Pascal implementation

The P6 Compiler

Scott A. Moore

[1 Overview of Pascal-P6 5](#_Toc527912156)

[1.1 Introduction 5](#_Toc527912157)

[2 Pascal-P6 vs. Pascal-P5 6](#_Toc527912158)

[3 Pascal-P6 as a strict ISO 7185 compiler 6](#_Toc527912159)

[4 Pascal-P6 vs. FPC (Free pascal) and GPC (Gnu Pascal), Borland, UCSD or other dialects 7](#_Toc527912160)

[5 The Pascal-P6 source language 7](#_Toc527912161)

[5.1 Features of Pascaline NOT implemented in Pascal-P6 8](#_Toc527912162)

[5.2 Local extensions to Pascaline in Pascal-p6 8](#_Toc527912163)

[5.2.1 Special field characters on write/writeln 9](#_Toc527912164)

[5.3 Pascal-P6 implementation details 9](#_Toc527912165)

[5.3.1 Header files 10](#_Toc527912166)

[5.3.2 Alternative header value input 10](#_Toc527912167)

[5.3.3 Modular structure 11](#_Toc527912168)

[6 Using Pascal-P6 11](#_Toc527912169)

[6.1 Configuring P6 11](#_Toc527912170)

[6.2 Compiling and running Pascal programs with P6 12](#_Toc527912171)

[6.2.1 Compiling with multiple modules 12](#_Toc527912172)

[6.2.2 Compiling on different run configurations 12](#_Toc527912173)

[6.3 Compiler options 12](#_Toc527912174)

[6.4 Other operations 15](#_Toc527912175)

[6.5 Reliance on Unix commands in the P5 toolset 15](#_Toc527912176)

[6.6 The “flip” command and line endings 16](#_Toc527912177)

[7 Building the Pascal-P5 system 16](#_Toc527912178)

[7.1 Compiling and running P5 with the GCC/I80386 backend 16](#_Toc527912179)

[7.2 Compiling and running P5 with an existing ISO 7185 compiler 17](#_Toc527912180)

[7.3 Evaluating an existing Pascal compiler using P5 18](#_Toc527912181)

[7.4 Notes on using existing compilers 19](#_Toc527912182)

[7.4.1 GPC 19](#_Toc527912183)

[7.4.1.1 GPC on Cygwin 19](#_Toc527912184)

[7.4.1.2 GPC for mingw 20](#_Toc527912185)

[8 Files in the P5 package 21](#_Toc527912186)

[8.1 Directory: gpc 24](#_Toc527912187)

[8.2 Directory: gpc/linux\_X86 24](#_Toc527912188)

[8.3 Directory: mac\_X86 24](#_Toc527912189)

[8.4 Directory: gpc/standard\_tests 24](#_Toc527912190)

[8.5 Directory: gpc/windows\_X86 24](#_Toc527912191)

[8.6 Directory: ip\_pascal 25](#_Toc527912192)

[8.7 Directory: ip\_pascal/standard\_tests 25](#_Toc527912193)

[8.8 Directory: ip\_pascal/windows\_X86 25](#_Toc527912194)

[8.9 Subdirectory: sample\_programs 25](#_Toc527912195)

[8.10 Directory: standard\_tests 26](#_Toc527912196)

[9 Differences between Pascal-P4 and Pascal-P5 26](#_Toc527912197)

[9.1 Viewing changes 26](#_Toc527912198)

[9.2 Notes about change descriptions 26](#_Toc527912199)

[10 The intermediate language 27](#_Toc527912200)

[10.1 Format of intermediate 27](#_Toc527912201)

[10.1.1 Label 28](#_Toc527912202)

[10.1.2 Source line marker 28](#_Toc527912203)

[10.2 Intermediate instruction set 28](#_Toc527912204)

[10.3 System calls 42](#_Toc527912205)

[11 Testing P5 49](#_Toc527912206)

[11.1 Running tests 50](#_Toc527912207)

[11.1.1 testprog 50](#_Toc527912208)

[11.1.2 Other tests 51](#_Toc527912209)

[11.1.3 Regression test 51](#_Toc527912210)

[11.2 Test types 51](#_Toc527912211)

[11.3 The Pascal acceptance test 51](#_Toc527912212)

[11.4 The Pascal rejection test 52](#_Toc527912213)

[11.4.1 List of tests 53](#_Toc527912214)

[11.4.1.1 Class 1: Syntatic errors 53](#_Toc527912215)

[11.4.1.2 Class 2: Semantic errors 59](#_Toc527912216)

[11.4.1.3 Class 3: Advanced error checking 62](#_Toc527912217)

[11.4.1.4 Class 4: Field checks 62](#_Toc527912218)

[11.4.2 Running the PRT and interpreting the results 62](#_Toc527912219)

[11.4.2.1 List of tests with no compile or runtime error. 62](#_Toc527912220)

[11.4.2.2 List of differences between compiler output and “gold” standard outputs. 63](#_Toc527912221)

[11.4.2.3 List of differences between runtime output and “gold” standard outputs. 63](#_Toc527912222)

[11.4.2.4 Collected compiler listings and runtime output of all tests. 63](#_Toc527912223)

[11.4.3 Overall interpretation of PRT results 63](#_Toc527912224)

[11.5 Sample program tests 63](#_Toc527912225)

[11.6 Previous Pascal-P versions test 64](#_Toc527912226)

[11.6.1 Compile and run Pascal-P2 64](#_Toc527912227)

[11.6.2 Compile and run Pascal-P4 64](#_Toc527912228)

[11.7 Self compile 65](#_Toc527912229)

[11.7.1 pcom 65](#_Toc527912230)

[11.7.1.1 Changes required 65](#_Toc527912231)

[11.7.2 pint 66](#_Toc527912232)

[12 pgen\_gcc\_i80386 67](#_Toc527912233)

[12.1 Generated assembly file conventions 68](#_Toc527912234)

[12.2 Labeling conventions 69](#_Toc527912235)

[12.3 Calling conventions 69](#_Toc527912236)

[12.4 clib interface library and support module 70](#_Toc527912237)

[12.5 Modifications required to pcom for pgen 70](#_Toc527912238)

[12.6 Overview of pgen\_gcc\_i80386 functionality 70](#_Toc527912239)

[12.6.1 Processing expressions 71](#_Toc527912240)

[12.6.2 Register allocation 72](#_Toc527912241)

# Overview of Pascal-P6

This section contains background material on Pascal-P6. If you want to get started using Pascal-P6 now, skip to “Using Pascal-P5”

## Introduction

The Pascal-P series compilers were the original proving compilers for the language Pascal. Created in 1973, Pascal-P was part of a “porting kit” designed to enable the quick implementation of a Pascal language compiler on new machines. It was released by Niklaus Wirth’s students at ETH in Zurich.

The implementation and description of the language Pascal in terms of itself and in terms of a “pseudo machine” were important factors in the propagation of the language Pascal. From the early version of Pascal-P came the CDC 6000 full compiler at Zurich, several independent compilers including an IBM-360 compiler and a PDP-11 compiler, and the UCSD “byte code” interpreter.

The original article for the Pascal-P compiler is at:

<http://www.standardpascal.org/The_Pascal_P_Compiler_implementation_notes.pdf>

In the name “Pascal-P” the “P” stood for “portable”, and this was what Pascal-P was designed to do. It also stood for an example and reference implementation of Pascal, although Wirth later issued a paper, together with Tony Hoare for the “Axiomatic definition of Pascal”, which was also aimed at exactly specifying the semantics of Pascal.

As the importance of Pascal-P grew, the authors adopted a version number system and working methodology for the system. A new, cleaner and more portable version of the system was created in 1974 with the name Pascal-P2, and left the multiple early versions of the system as termed Pascal-P1.

From the Pascal-P2 revision of the compiler comes many of the original Pascal compilers, including UCSD. In 1976, Wirth’s group made one last series of improvements and termed the results Pascal-P3 and Pascal-P4. Pascal-P3 was a redesigned compiler, but used the same pseudo machine instruction set as P2, and thus could be bootstrapped from an existing P2 implementation. P4 featured a new pseudo instruction set, and thus was a fully redesigned compiler.

Pascal-P was always an imcomplete implementation of the Pascal language (a subset), and was designed to be so. After it was created, the ISO 7185 standard for Pascal was issued, and today Pascal-P4 exists and is still usable with minor changes to bring it into ISO 7185 compliance.

However, Pascal-P4 has it’s legacy problem of being a subset compiler of the full language. Further, it is only usable for programs that avoid its weaknesses, such as string storage. Keep in mind that Pascal-P was never designed to be a general purpose system, but rather to compile itself on a new machine – and then rapidly be improved to become a full compiler.

Inn 2008 I set out to improve the P4 code to accept the full language Pascal as stated by the ISO 7185 language standard. The name of the result was obvious: Pascal-P5, although I certainly beg forgiveness of any of the original Wirth group members who might be offended by my implication of extending and improving their work. Think of it this way: I too have been a lifelong student of Niklaus Wirth’s, abet a long distance one.

Pascal-P5 exists as a separate project and is an interpreter only. It can form the basis for any ISO 7185 based project. The project intent was to only add the minimum code to the Pascal-P4 implementation to bring it up to ISO 7185 status. However, it also introduced a large suite of tests to verify compliance with the ISO 7185 Pascal standard.

The purpose of Pascal-P6 was to create a working compiler implementation from Pascal-P5. Thus Pascal-P5 serves as a working, interpretive model of ISO 7185 Pascal, and Pascal-P6 serves as an actual machine implementation of it.

The first task of the Pascal-P6 implementation was to create a series of extensions to ISO 7185 Pascal. All actual machine implementations of Pascal, including the original CDC 6600 implementation, include a series of extensions. With Pascal-P6, I wanted to introduce a series of extensions that would both bring the compiler to a usable state as practical compiler and also bring it up to the “state of the art” (at least as of 2018).

I realize that introducing a set of new features to Pascal is going to be controversial. I wanted the extensions to be more than just a laundry list of features added by a compiler implementor (as has happened with Pascal so many times in the past). Thus I collected the features implemented by Pascal-P6 as a formal language in its own right, as the language “Pascaline”. There is a formal specification accompanying Pascal-P6 (“pascaline.docx”) as well as a rationale (“Pascaline\_rationale.docx”), a discussion of both the language, its features, and its implementation.

# Pascal-P6 vs. Pascal-P5

P6 processes the entire ISO 7185 Pascal language as a proper subset of Pascaline. At the ISO 7185 Pascal level, Pascal-P6 and Pascal-P5 are completely compatible. However, there are reasons you should use Pascal-P6, even if you are only using strictly the ISO 7185 Pascal language:

* Pascal-P6 provides many new features, including debug mode, enhanced standards checking, and new output modes such as the faster cmach mode and eventually target machine output generators.
* Pascal-P6 fixes some bugs/issues with Pascal-P5. Just as Pascal-P4 fixed some Pascal-P2 issues, and Pascal-P5 fixed some Pascal-P4 issues, not all fixes of Pascal-P5 problems in Pascal-P6 made it back to Pascal-P5. There is considerable work involved with backwards porting all such fixes.

Thus unless you have a good reason, you should be using Pascal-P6, the latest in the series. See also the comments on strict ISO 7185 useage in the next section.

# Pascal-P6 as a strict ISO 7185 compiler

Pascal-P6 can easily be used as a strict ISO 7185 Pascal compiler. Using the s+ flag, or “strict” flag, the source language for Pascal-P6 and Pascal-P5 is identical. It is, however, important to understand the exact meaning of the s+ flag.

I have used the term “strict” for the s+ flag, and not “standard”, because the P6 compiler is fully compatible with the ISO 7185 Pascal language regardless of if the flag is off or on[[1]](#footnote-1). In pascal-P6 the option has the following meanings when ON:

* All extended keywords from Pascaline are turned OFF, meaning they can be used as program identifiers.
* So called “force sequences” or “\” character special interpretations in strings is turned off, meaning the “\” character has no special meaning in strings.

Both of these are in compliance with the ISO 7185 standard. An implementation is allowed to define new keywords in ISO 7185 Pascal. Also the interpretation of characters in ISO 7185 Pascal is up to the implementation (it does not even specify ASCII).

I recommend that even if you are developing strict ISO 7185 Pascal source code that you leave the s flag OFF (the default). Why? If the flag is off, that allows you to use keywords that are reserved in Pascaline, or use the force character “\” in your source. This will cause problems later if you then decide to use Pascaline extensions. In fact, the best way to develop truly portable ISO 7185 Pascal programs is to avoid using other ISO 7185 Pascal implementations’ keywords as well. You might want to examine the keyword list for FPC (Free Pascal) and GPC (Gnu Pascal) as well to make sure.

The use of the force character (“\”) is trickier. If you put it in your ISO 7185 programs with the s flag OFF (s-), then you will need to specify “\\” to get just the force character. But if you then compile it on a system that does not understand force sequences, it will evaluate to “\\” (double “\”). The best method is to avoid the force character whenever possible. Fortunately, most ISO 7185 Pascal compilers today do understand force sequences.

When using Pascal-P6 as a ISO 7185 compiler, I recommend that you occasionally check your compiles by turning on the s+ switch. This will make sure that you have not inadvertently used Pascaline constructs in your source.

# Pascal-P6 vs. FPC (Free pascal) and GPC (Gnu Pascal), Borland, UCSD or other dialects

Pascal-P6 does not accept the common dialects[[2]](#footnote-2) of Borland or UCSD languages[[3]](#footnote-3). There are good implementations of these dialects, both by Borland themselves and by the FPC group. My recommendation is if you want to use that source language, use the best compiler for it, namely the FPC or Borland compilers. There is no plan for a Pascal-P series compiler that is also Borland dialect compatible (perhaps you could make one?), and further, the type protection model of the two are diametrically opposed.

FPC recently completed implementation of an ISO 7185 compliance switch. This makes it a good backup compiler for ISO 7185 Pascal compliant applications. See the FPC web site for more details.

Both UCSD, variants of (like Apple Pascal) and GPC are falling out of use and support or have fallen out of use and support.

See also then next section for comments on source language.

# The Pascal-P6 source language

Pascal-P6 understands the Pascaline source language as documented in “pascaline.docx”, available in the doc or documents directory of Pascal-P6 project. Pascaline is fully compatible with ISO 7185. There are, however, several things you will find only in this documentation and not the Pascaline specific documentation:

* Features of Pascaline that are not (yet) implemented in Pascal-P6.
* Local extensions in Pascal-P6 that are not also in general Pascaline.
* Concrete details of features in Pascaline specific to Pascal-P6.

These will be detailed.

## Features of Pascaline NOT implemented in Pascal-P6

The following features of Pascaline are not yet implemented in Pascal. The expectation is all of them will be implemented shortly. Also keep in mind that Pascal-P6 is still a project in development, and carries a version number < 1.0.

|  |  |
| --- | --- |
| Pascaline manual section | Description |
| 6.18 | Fixed types |
| 6.23 | Extended function results |
| 6.25 | Overloading of procedures and functions |
| 6.26 | Operator overloads |
| 6.35 | Matrix mathematics |
| 6.36 | Properties |
| 6.38 | Channels |
| 6.39 | Classes |

Note: outside of 6.18 Fixed types, and 6.25 Overloading of procedures and functions, Pascal-P6 implements all of the IP Pascal source language.

In addition to the list, Pascal-P6 does not implement:

* Parallel modules.
* Most of the annexes, including Unicode, string library, system library, etc.

Additionally, I plan to implement a full unsigned cardinal type. Presently, it is just an alias of integer, as the Pascaline standard allows for.

Please see “pascaline.docx” in the doc file section for more details on the Pascaline language.

## Local extensions to Pascaline in Pascal-p6

Radix markers on write/writeln

Pascal-P6 will take radix characters on output integer or integer compatible values, consisting of:

|  |  |
| --- | --- |
| Radix character | Meaning |
| $ | Hexadecimal (base 16) |
| % | Binary (base 2) |
| & | Octal (base 8) |

Example:

writeln(456$);

Writes “1c8”, the hexadecimal equivalent of 456 decimal. The way to remember this is: if the radix symbol preceeds the number, it means convert ***from*** the radix, if the radix symbol follows the number, it means to convert ***to*** the radix.

### Special field characters on write/writeln

|  |  |
| --- | --- |
| Field character | Meaning |
| \* | On string type only, means to interpret the string as padded string. |
| # | On integers, means to preserve leading zeros. |

For the “\*” field character, this means to take the field length as the part of the given string without trailing spaces, i.e.:

writeln(‘my dog has fleas ‘:\*);

Means to treat this output as equivalent to:

writeln(‘my dog has fleas ‘:16);

Or to automatically set the length according to the right space padded content of a string.

This special field character ***does not*** allow for a field value to be parsed.

For the “#” field character, it means to fill the field with leading zeros:

writeln(123:#10);

Would print:

0000000123

This special field character ***does*** require a field value to follow.

Special field characters on read/readln

Field character Meaning

\* Interpret strings read as right space padded strings

For the special character “\*”, read/readln interprets this to mean to fill out the provided string if eoln() is encountered. This allows any length of string to be read, and then padded out to the end of the string.

## Pascal-P6 implementation details

There are no particular limits of integer sizes, reals or sets built into Pascal-P6. As a hosted compiler, it picks up many or even most of its characteristics from the host compiler.

At this writing, there is a 32 bit and a 64 bit implementation of Pascal-P6. The following characteristics exist:

|  |  |
| --- | --- |
| Type | Size/limit |
| integer | maxint = 2147483647 |
| linteger | Alias of integer |
| cardinal | Alias of integer |
| lcardinal | Alias of integer |
| set | 256 elements |
| real | 64 bit real |
| sreal | Alias of real |
| lreal | Alias of real |

The maximum size of any input line, identifier, or string constant is 250 characters.

The base character set is ISO 8859-1, or 8 bit characters. The compiler has no reliance on any character in the range 128 to 255 and will accept any character in that range for input or output. In the characters 0 to 128,

### Header files

All of the Pascaline header files are implemented, input, output, error, list and command. In the interpreters pint, pmach, and cmach, output, error and list all go to the standard output file. As detailed in Annex C of the Pascaline standard, undefined program header parameters must be declared for type. Any type of valid Pascaline file is acceptable, including complex structured types.

The filenames corresponding to the names of the header file parameters are read from the command line:

program print(myfile, output);

var myfile: text;

…

executed with a command line of:

print document.txt

Will assign file myfile to the name on disk “document.txt”.

The command file reads the command line passed to the program on startup. There is only one line in the file, and eof() immediately follows eoln() in the file. When the command file is read as well as other parameters appear in the program header, then the command file will be positioned after all other header file names or entries have been read from the command line.

### Alternative header value input

Pascal-P6 can have integer or real parameters in the header file, in addition to just files:

program test(myint, myreal);

var myint: integer;

myreal: real;

Pascal-P6 automatically reads these values off of the command line and into the specified variables. If a they are mixed values on the header, they will be read in turn off the command line as specified in Annex C of the Pascaline standard.

### Modular structure

Pascal-P6 implements modules by outputting an intermediate file for each module, then the intermediates are concatenated and loaded into pint as a single file. The intermediate contains sufficient information for pint to understand the module structure, and it links the modules together.

Modules in Pascal-P6 are implemented by the following method:

1. Each module calls its own initializer block, then calls the next module in the stacking order. When the module it calls returns, that module then calls the deinitializer for the module, and returns to the module that called it. Program modules only have a single initializer block. When the block is done, the program module returns.
2. Each module is stacked up on the last. The interpreter or runtime system arranges to call the first module in the stacking order, then when it returns, the entire program exits.
3. Its up to the user to stack the modules properly on the command line. If a module is called via a function or procedure, and accesses global data from that module, the program will fault with uninitialized access error. Module overrides are an exception. Each virtual function or procedure in the module is initialized to point to an error routine when the module is started, and attempts to override an uninitialized virtual vector will generate an error.
4. The program module MUST be the last module in the stacking order.
5. A process module is like a program module, but it starts a new processor thread to run its initialize block, then calls the next module in the stacking order. From the time the thread is branched off to run the process block, the main thread and the thread running the process module run in parallel.

# Using Pascal-P6

## Configuring P6

P6 has a simple configuration script to set up the binary, script files and compiler in use for the system, that uses the proper defaults for your system:

[Windows]

> setpath

> configure

> make

[Linux/Mac]

$ ./setpath

$ ./configure

$ make

You can avoid “setpath” by placing the ./bin directory on your path.

The configure script attempts to automatically determine the environment you are running under, choose the correct compiler, bit width of your computer, etc. You can override this by using the options for configure:

|  |  |
| --- | --- |
| Option | Meaning |
| --gpc | Selects the GPC compiler. |
| --ip\_pascal | Selects the IP Pascal compiler. |
| --fpc | Selects the FPC compiler. |
| --32 | Selects 32 bit mode. |
| --64 | Selects 64 bit mode. |
| --help | Prints a help menu. |

The configure script will take the preconfigured versions of the p6 binaries, the script files and other files and install them for the specified compiler.

Although the directory will contain working copies of the binaries and scripts, there is no guarantee which version it contains. Always run the configure script to setup the particular system you are using.

## Compiling and running Pascal programs with P6

To simply compile a run a program, use the P6 batch file:

C:\> p6 hello

When a pascal program is run this way, it gets it's input from the terminal (you), and prints its results there. The p6 script accommodates the compiler that was used to build the system, and therefore you don’t need to know the exact command format of the executable.

### Compiling with multiple modules

The P6 script can accept multiple modules:

C:\> p6 mylib1 mylib2 test

The script will compile and load each of the modules in turn, and run them as a group. The program module should always be last in the module stacking order. See xx.

### Compiling on different run configurations

There are several types of run arrangements for P6. The current options are:

|  |  |
| --- | --- |
| Option | Meaning |
| --pmach | Run on the pmach machine. |
| --cmach | Run on the cmach machine. |
| --package | Run via the packaging option (using cmach). |

Note that the result of running in these different configurations will not produce different results, and this feature is mostly used for testing the compiler.

## Compiler options

P6 uses a "compiler comment" to indicate options to the compiler, of the form:

(\*$option+/-,...\*)

This option can appear anywhere a normal comment can. The first character of the comment MUST be "$". This is followed by any number if option switches separated by ",". If the option end with "+", it means to turn it on. If the option ends with "-", it means turn it off.

Example:

(\*$l-\*)

Turns the listing of the source code OFF.

The following options are available:

|  |  |  |
| --- | --- | --- |
| Option | Meaning | Default |
| t+/- | Print/don't print internal tables after each routine is compiled. | OFF |
| l+/- | List/don't list the source program during compilation. | ON |
| d+/- | Add extra code to check array bounds, subranges, etc. | ON |
| c+/- | Output/don't output intermediate code. | ON |
| v+/- | Check variant records. | ON |
| r+/+ | Perform reference checking. | ON |
| u+/- | Perform undefined variable checking. | ON |
| s+/- | Restrict input language to ISO 7185 Pascal. | OFF |
| x+/- | Dump lexical information during the run. | OFF |
| b+/- | Print goto labels used at the end of the compile. | OFF |
| y+/- | Dump display information at the end of each block (symbols). | OFF |
| i+/- | Check VAR block violations. | OFF |
| g+/- | Dump goto and other location labels at intermediate assembly time. | OFF |
| h+/- | Add source line sets to code. This tells pint where the source lines are in the code. | ON |
| n+/- | Obey heap recycle requests. If false, no space is recycled. |  |
| o+/- | Check for arithmetic overflow. |  |
| p+/- | Check for reuse of freed entry in heap space. |  |
| m+/- | Break returned entries in heap into occupied and free blocks. |  |
| q+/- | Check accesses to undefined memory. |  |
| w+/- | Enter debug mode (pint only). |  |
| r+/- | Perform reference checks. |  |
| f+/- | Perform source level debugging. |  |
| e+/- | Output binary deck from pint instead of running the assembled results (pint only). |  |

It is possible to cause problems in the code if the options are turned on or off arbitrarily in the middle of the program. It is recommended that you place option sets at the top of the program only unless you are sure what you are doing.

P6 provides a lot of switches for developers to use to debug the compiler. A good rule is: if you don’t know what a switch does, don’t use it. P6 is a compiler developer’s compiler. This means that everything in the system is available to change, including things that make no sense for end users.

Note that a lot of the debug switches previously available in Pascal-P5 have been moved to the debug mode in P6. See xx.

## Other operations

Within the P6 toolset, you will find a series of scripts to perform common operations using P6. This includes building the compiler and interpreter using an existing ISO 7185 compatible compiler, and also testing P6.

The scripts used in P6 are designed to be independent of what operating system you are running on. The P6 system as been successfully run on the following systems:

* Windows
* Ubuntu linux
* Mac OS X

To enable this to work, there are two kinds of scripts available, one for DOS/Windows command shells, and another for Unix/Bash. These two script files live side by side, because the DOS/Windows scripts use a .bat extension, and Bash scripts use no extentions. Thus, when a script command is specified here, the particular type of script file is selected automatically.

The only exception to this rule is that Unix users commonly do not place the current directory in the path. This means to execute a script file in the current directory, you need to specify the current directory in front of the script. For example:

~/P6$ ./P6 hello

## Reliance on Unix commands in the P6 toolset

Most of the scripts in this package, even the DOS/Windows scripts, rely on Unix commands like cp, sed, diff, chmod and others. I needed a reasonable set of support tools that were command line callable, and these are all both standard and reasonable.

For Windows, the Cygwin toolset is available:

http://www.cygwin.com

Note that to run the cygwin tools, you will need the environment variable:

CYGWIN=nodosfilewarning

This prevents cygwin utilities from complaining about dos mode file specifications.

An alternative to Cygwin is the Mingw toolkit. Mingw uses GNU programs that are compiled as native Windows .exe files without special .dll files. It typically has better integration with Windows than Cygwin, since it does not try to emulate Unix on Windows.

Where possible, I have tried to use DOS/Windows commands. The scripts are available in both DOS/Windows and bash versions. I could have just required theuse of bash, which is part of the cygwin toolkit, but my aim is not to force Windows users into a Unix environment.

## The “flip” command and line endings

Every effort was made to make the Pascal-P6 compile and evaluate system idenpendent of what system it is running on, from Windows command shell, to Linux with Bash shell. One common thing I have found is that several utilities don’t appreciate seeing a line ending outside of their “native” line ending, such as CRLF for Windows, and LF for linux. Examples include “diff” (find file differences) and Bash.

Therefore many of the scripts try to remove the line ending conserations, either by ignoring such line endings, or by converting all of the required files to the particular line ending in use.

The key to this is the “flip” utility. After searching for several line ending converters, “flip” was found on the most number of systems, as well as being one of the most clear and reliable utilities (it translates in both directions, it tolerates any mode of line ending as input, will not corrupt binaries, etc.).

Unfortunately, even flip was not found on some systems. The simpliest way to fix this was to include the flip.c program with the distribution, then let you compile to form a binary on your system to replace the utility.

To make the flip utility, you run:

$ make\_flip

Then flip will exist in the root directory.

Note that if you are using SVN to retrieve the P6 project, the file entries are given the “OS specific” line ending property. This means both that the line endings will be converted to the line endings particular to your OS, and also prevents line endings from causing SVN to think the file has changed.

# Pascal-P6 implementation language

Pascal-P6 accepts the language Pascaline. However, Pascal-P6 is itself written in ISO 7185 Pascal. This means that Pascal-P6 can be run on any host compiler that complies with the ISO 7185 Pascal language, but then itself implements the full Pascaline language. Thus it can be used to “bootstrap” any ISO 7185 Pascal installation into the Pascaline language.

To do this, it utilizes two basic “tricks” to get beyond the limitations in ISO 7185 Pascal. First, Pascal-P6 uses the Unix/Linux cpp preprocessor to inable things like file includes and conditional compilation. cpp is run via a script pascpp or pascpp.bat. It takes a \*.pas pascal input source and creates a file by the name \*.mpp.pas, or the processed output file, then it uses the ISO 7185 Pascal host compiler to compile that file.

To make cpp work on a non-C based language, the options:

cpp -P -nostdinc -traditional-cpp

Are used. This makes cpp run as close to as possible to not relying on any interpretation of the source language.

If you are going to make changes and compile the compiler, you should expect that if you get errors, you need to look in the \*.mpp.pas files, not the original \*.pas files. The reason is because while C compilers can compensate for the line and character location differences of error reporting in the source file, Pascal compilers (esp. Pascal-P6) cannot do that. Thus you must examine the cpp output files to make sense of the errors encountered.

## Language extension routines

Although it may seem strange that a limited compiler as an ISO 7185 Pascal source compiler can compile for a language much more powerful than its source language, in fact it is not difficult and is done commonly. The biggest difference is in fundamental operating system features covered by the language. Thus, access to named files, seeking within those files, and access to the command line passed to the program are common functions that are beyond the ISO 7185 Pascal language. These are handled by so called “language extension routines”. These are short routines with standard ISO 7185 call sequences that implement the extended feature. In fact they are (at current time):

procedure assigntext(var f: text; var fn: filnam);

Assign name to text file.

procedure assignbin(var f: bytfil; var fn: filnam);

Assign name to binary file.

procedure closetext(var f: text);

Close text file.

procedure closebin(var f: bytfil);

Close binary file.

function lengthbin(var f: bytfil): integer;

Find length binary file.

function locationbin(var f: bytfil): integer;

Find location binary file.

procedure positionbin(var f: bytfil; p: integer);

Position binary file.

procedure updatebin(var f: bytfil);

Update binary file.

procedure appendtext(var f: text);

Append text file.

procedure appendbin(var f: bytfil);

Append binary file.

function existsfile(var fn: filnam): boolean;

Find file exists by name.

procedure deletefile(var fn: filnam);

Delete file by name.

procedure changefile(var fnd, fns: filnam);

Change filename.

procedure getcommandline(var cb: cmdbuf; var l: cmdnum);

Get the shell command line.

For each different host compiler, a complete library implementing these calls in the particular host language exists. The following are the host support library files at this writing:

extend\_gnu\_pascal.inc

Implements specific extension routines for GPC (Gnu Pascal) as the host compiler.

extend\_iso7185\_pascal.inc

This file is a dummy, and simply gives an error when any of the routines are called. This allows Pascal-P6 to be compiled with a strict ISO 7185 compiler (like Pascal-P5), and still function, but without the extended capabilities.

extend\_pascaline.inc

Implements the extension routines using Pascaline extensions. This file is used when there is a native Pascaline implementation to simply self define the routines.

# Building the Pascal-P6 system

## Compiling and running P6 with an existing ISO 7185 compiler

You do not need to compile P6 unless you are using an alternative compiler or installation. The current P6 has been compiled and run with the following compilers and operating systems:

|  |  |
| --- | --- |
| Compiler | Installations |
| IP Pascal | Windows |
| GPC | Windows, Ubuntu, Mac OSx |

First, you must have a ISO 7185 Pascal compiler available. There are several such compilers, see:

http://www.standardpascal.org/compiler.html

You will probally need to compile pcom.pas and pint.pas with the ISO 7185 Pascal compatibility mode option on for your compiler. See your documentation for details.

If you are using a compiler or version of a compiler that is not tested to ISO 7185 standards, you will want to make sure that it is ISO 7185 compliant. See “Tesing P5” section 18 for details on how to test an existing compiler to ISO 7185 standards.

To compile pcom and pint, the components of the P6 compiler, use the script files:

cpcom Compile pcom to binary with a specific compiler.

cpint Compile pint to binary with a specific compiler.

For convienence, both of these are run by the command:

Build Compile pcom and pint.

cpcom and cpint are specific to the compiler you are using. You can use the script configure to select which existing compiler to run:

configure ip\_pascal

Or

configure gpc

To run the other programs and batch files, you should modify the following files to work with your compiler:

p6.bat The single program compile and run batch file.

compile.bat To compile a file with all inputs and outputs specified.

run.bat To run (interpret) the intermediate file with all inputs and outputs specified.

The reason you need to change these files is because pcom.pas uses the header file "prr" to output intermediate code, and pint.pas uses "prd" for input and "prd" for output. You need to find out how to connect these files in the program header to external named files.

For example, in IP Pascal, header files that don't bear a standard system name (like "input" and "output") are simply assigned in order from the command line. Thus, P6.bat is simply:

pcom %1.p6 < %1.pas

pint %1.p6 %1.out

Where %1 is the first parameter from the command line.

P6.bat lets the input and output from the running program go to the user terminal. Compile.bat and run.bat both specify all of the input, output, prd and prr files. The reason the second files are needed is so that the advanced automated tests can be run using batch files that aren't dependent on what compiler you are using.

If your compiler does nothing with header files at all, you will probally have to change the handling of the prd and prr files to get them connected to external files. To do this, search pcom and pint for "!!!" (three exclamation marks). This will appear in comments just before the declaration, reset and rewrite of these files.

## Evaluating an existing Pascal compiler using P6

If you plan to compile and run P6 using your compiler, you should evaluate your compiler's ISO 7185 Pascal compliance. Of course, simply compiling pcom.pas and pint.pas is one way to achieve that. But since this package gives you the ability to fully evaluate your compiler, I would suggest you use it.

First, you need to determine if your compiler has a ISO 7185 Pascal compliance option and turn it on if needed. I say "if needed", because some compilers actually change their behavior with the option enabled, and thus it is not possible to compile and run standard Pascal programs unless the option is on (a very unfortunate property of a Pascal implementation).

Within ISO 7185 Pascal, there are two characteristics of an implementation that could cause P6 to not compile, even if the implementation otherwise completely complies with the standard:

1. Conflict with extended keywords.
2. Character formats.

The first concerns an implementation that defines a new keyword conflicting with an identifier used in P6. For example, if your compiler has an extended keyword "variant", this would cause pcom.pas not to compile: it uses that as an identifier. Ideally, the ISO 7185 Pascal option should turn off such extended

keywords, but you may have to invoke another option to do this. Such extended keywords are allowed by the ISO 7185 Pascal standard.

The second is simply that the character set in use is not specified by the ISO 7185 Pascal standard. This is rarely an issue now, because virtually all implementations are based on either ISO 8859-1 (or ASCII), or are based on a character set that contains ISO 8859-1 as a base standard (both ISO 8859 and Unicode do this).

It is also possible that an implementation may define special character formats. For example, the commonly implemented character force sequences are a special format:

'this is a string\n'

This is valid, since ISO 7185 Pascal does not specify the exact format of strings.

Fortunately, P6 does not contain nor need force sequences, so this will not cause problems.

Besides compiling pcom.pas and pint.pas, I strongly recommend you run and check at least ISO7185pat.pas. This is a fairly comprehensive test of ISO 7185 Pascal compliance.

If you wish to run the entire compliance test on your compiler, you simply need to change or create a version of compile.bat and run.bat for your implementation that operate with your compiler. Then you can run the regression test ***without*** the self compile features (cpcoms and cpints).

Finally, building P6, and then running it through a full regression is itself a good final test of ISO 7185 compliance. It does not substitute for direct testing of your compiler. P6 could well run correctly even if your compiler is not fully ISO 7185 Pascal compliant!

For further details concerning the ISO 7185 tests, see 18 "Tesing P5”.

## Notes on using existing compilers

### GPC

GPC (GNU Pascal Compiler) is used in the following version:

GNU Pascal version 20070904, based on gcc-4.1.3 20080704 (prerelease) (Ubuntu 2.1-4.1.2-27ubuntu2).

Copyright (C) 1987-2006 Free Software Foundation, Inc.

I have had several difficulties with other versions of GPC, which give errors on standard ISO 7185 source, or crash, or other difficulties. The GPC developers announced they were halting development on GPC in the gpc mailing list. Please see their web page:

<http://www.gnu-pascal.de>

For any further information.

The main difficulty with GPC vis-a-vie P6 is that testing of the GPC compiler for ISO 7185 compatability was not regularly done on GPC releases. Thus, otherwise working GPC releases were not able to compile and run standard ISO 7185 source code.

Because of this, I can only recommend the above version of GPC be used, which compiles and runs P6 error free.

In addition, please be aware that I have not run the GPC compiler, including the above version, through a current ISO 7185 compliance test such as appears here. My only concern is that GPC be able to complile and run P6, and that the resulting P6 runs the compliance tests. I leave it for others to run full compliance for GPC itself.

#### GPC on Cygwin

The current Cygwin release as of 2012/03/26 does not work, since it uses GPC 2005, and is broken at that (it has the .dlls for GPC installed incorrectly).

A procedure to use GPC under the current Cygwin I have used is as follows:

1. Install the latest version of Cygwin (the one I tried is Cygwin/X, a very useful package).
2. Place c:\cygwin\bin on your path.
3. Go to the website:

<http://www.gnu-pascal.de/binary/cygwin/>

And download and install:

[gpc-20070904-with-gcc.i686-pc-cygwin.tar.gz](http://gnu-pascal.de/contrib/chief/win32/cygwin/gpc-20070904-with-gcc.i686-pc-cygwin.tar.gz)

(4.4mb, gpc-20070904, based on gcc-3.4.4, with gcc-3.4.4 support files)

1. After installing this package in an appropriate directory, say c:\gpc, modify your path to include c:\gpc\usr\bin ahead of the c:\cygwin\bin directory in the path.
2. Add cygwin=nodosfilewarning (as stated in section 15 “Reliance on Unix commands in the P5 toolset”).

Now you will be able to follow the normal gpc instructions here to get P6 running, using the standard Windows command shell. Note that this trick won’t work with the command shells Cygwin provides.

To reiterate the steps that follow:

$ configure gpc Configure for GPC compiler.

$ build Build the P6 binaries.

$ regress Run the regression suites to check the P6 compiler.

#### GPC for mingw

Mingw (Minimal GNU for Windows) is a different port of the GNU catalog for windows that runs directly on windows. That is, each binary is statically linked with its support library, and it is designed to work with windows directly.

As Cygwin has become more and more a full emulation of the Unix environment (a good thing), it has become less usable in interaction with other Windows programs. Thus I have found the mingw package more cooperative for every day Windows work.

Mingw does not come natively with GPC installed (or much else). I recommend you also pick up the MSYS package for mingw, which is a series of GNU programs that are compiled to run in the windows environment using Mingw.

To get the mingw distribution of GPC, follow the steps:

1. Go to the website:

<http://www.gnu-pascal.de/binary/mingw32/>

And download and install:

[gpc-20070904-with-gcc.i386-pc-mingw32.tar.gz](http://gnu-pascal.de/contrib/chief/win32/cygwin/gpc-20070904-with-gcc.i686-pc-cygwin.tar.gz)

(4.4mb, gpc-20070904, based on gcc-3.4.5, with gcc-3.4.5 support files)

1. After installing this package in an appropriate directory, say c:\gpc, modify your path to include c:\gpc\usr\bin directory in the path.

Note that this is based on a slightly different version than Cygwin.

To reiterate the steps that follow:

$ make\_flip Create flip.exe because msys does not have one. Note that you will need to move the resulting flip.exe to your bin directory.

$ configure gpc Configure for GPC compiler.

$ build Build the P6 binaries.

$ regress Run the regression suites to check the P6 compiler.

Note: The bash command shell does not work with programs generated by GPC. It gives and error when executing them. Thus it is necessary to use the standard command shell in conjuction with Mingw utilities.

# Files in the P6 package

Note: for script files, both a DOS/Windows (X.bat) and bash script (X) are provided. Their function is identical, one is for use with the DOS/Windows command shell, the other for bash shell.

build.bat

build Build the compiler and interpreter. Just runs cpcom and cpint.

clean.bat

clean Cleans generated and temp files out of the directories. This is used, for example, before generating a .zip file version of the system.

compile

compile.bat Batch mode compile for P6. It takes all input and output from supplied files, and is used by all of the other testing scripts below. You will need to change this to fit your particular Pascal implementation.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

configure.bat

configure Sets the current compiler to use to create P6 binaries.

cpcom

cpcom.bat Script file to compile the compiler side of P6 to binary on your computer.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

cpcoms

cpcoms.bat Self compile, run and check the pcom.pas file. This batch file compiles com.pas, then runs it on the interpreter and self compiles it, and checks the intermediate files match.

cpint

cpint.bat Script file to compile the interpreter side of P6 to binary on your computer.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

cpints

cpints.bat Self compile, run and check the pint.pas file. This batch file compiles pint.pas and iso7185pat.pas, then runs pint on itself and then runs iso7185pat.pas, and checks the result file.

diffnole

diffnole.bat Runs a diff, but ignoring line endings (DOS/Windows vs. Unix).

doseol

doseol.bat Fixes the line endings on text files to match the DOS/Windows convention, CRLF.

fixeol

fixeol.bat Arranges the line endings on bash scripts to be Unix, and those of the DOS/Windows scripts to be DOS/Windows line endings. This is required because the editors on the respective systems insert their own line endings according to system, and this can cause problems when they are run on a different system.

flip.c C program to replace the local version of “flip”, the Unix line ending fixup tool. It is provided in source form here because not all Unix installations have it (for example MAC OS X didn’t have it). This allows you to compile it yourself for your target system.

iso7185.html

iso7185.pdf The full ISO 7185 Pascal standard, in html and pdf forms.

iso7185rules.html A description of the ISO 7185 Pascal language.

make\_flip

make\_flip.bat A script to compile deoln and ueoln and create a flip script for Unix. This is used to replace the “flip” program if required.

news.txt Contains various information about the current release.

P6

P6.bat A batch file that compiles and runs a single Pascal program. You will need to change this to fit your particular Pascal implementation. It uses input and output from the terminal, so it is a good way to run arbitrary programs.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

pcom

pcom.exe The IP Pascal compiled pcom binary for Windows/Unix. See comments in 12 “Compiling and running Pascal programs with P5” for how to use this. All of the supplied batch files are customized for this version.

pcom.pas The compiler source in Pascal.

pint.exe The IP Pascal compiled pint binary for Windows. See comments in 12 “Compiling and running Pascal programs with P5” for how to use this. All of the supplied batch files are customized for this version.

pint.pas The interpreter source in Pascal

prtprt.bat Batch file, takes all of the rejection test error files and concatenates them to standard\_tests/iso7185prt.lst.

readme.txt Brief introduction to the project, it points to this document now.

regress

regress.bat The regression test simply runs all of the possible tests through P6. It is usually run after a new compile of P6, or any changes made to P6.

run

run.bat Batch mode run for P6. It takes all input and output from supplied files, and is used by all of the other testing scripts below. You will need to change this to fit your particular Pascal implementation.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

testpascals

testpascals.bat Runs a compile and check on pascals.pas, the Pascal subset interpreter created by Niklaus Wirth.

testprog

testprog.bat An automated testing batch file. Runs a given program with the input file, delivering an output file, then compares to a reference file.

Testprog is used to test the following program files for P6: hello, roman, match, startrek, basics and iso7185pat.

the\_P6\_compiler.doc

the\_P6\_compiler.docx

the\_P6\_compiler.html

the\_P6\_compiler.pdf This document in various forms, word 2007, word 1997, PDF, and HTML.

The\_Programming\_Language\_Pascal\_1973.pdf

Niklaus Wirth's description of the Pascal language, the last version to come from ETH. This is the equivalent of the "Report", from "Pascal user's manual and report [Jensen and Wirth].

todo.txt Contain a list of "to do" items in P6.

unixeol

unixeol.bat Fixes the line endings on text files to match the Unix convention, LF.

ZipP6.bat Creates a zipfile for the entire P6 project. This is used to create the releases available on the P6 web site.

## Directory: gpc

This directory contains scripts specifically modified for GPC.

compile

compile.bat The GPC specific version of the compile script.

cpcom

cpcom.bat The GPC specific version of the compile compiler script.

cpint

cpint.bat The GPC specific version of the compile interpreter script.

P6

P6.bat The GPC specific version of the P6 script.

run

run.bat The GPC specific version of the run script.

## Directory: gpc/linux\_X86

pcom

pint Contains binaries compiled by GPC for Linux/Ubuntu

## Directory: mac\_X86

A placeholder for Mac OS X binaries.

## Directory: gpc/standard\_tests

iso7185pat.cmp Contains the compare file for iso7185pat for gpc.

iso7185pats.cmp Contains the compare file for iso7185pats for gpc.

## Directory: gpc/windows\_X86

pcom.exe

pint.exe Contains binaries compiled by GPC for Windows.

## Directory: ip\_pascal

This directory contains scripts specifically modified for IP Pascal.

compile

compile.bat The IP Pascal specific version of the compile script.

cpcom

cpcom.bat The IP Pascal specific version of the compile compiler script.

cpint

cpint.bat The IP Pascal specific version of the compile interpreter script.

P6

P6.bat The IP Pascal specific version of the P6 script.

run.bat

run The IP Pascal specific version of the run script.

## Directory: ip\_pascal/standard\_tests

iso7185pat.cmp Contains the compare file for iso7185pat for IP Pascal.

iso7185pats.cmp Contains the compare file for iso7185pats for IP Pascal.

## Directory: ip\_pascal/windows\_X86

pcom.exe

pint.exe Contains binaries compiled by IP Pascal for Windows

## Subdirectory: sample\_programs

basics.pas A tiny basic interpreter in Pascal.

basics.inp Input test file for basics. In fact, it is a basic verion of "match" above.

basics.cmp Compare file for basics.

hello.pas One of several test programs used to prove the P6 system. This is the standard "hello, world" program.

hello.inp Input to hello for automated testing.

hello.cmp Hello compare file for automated testing.

match.bas The basic version of the match game.

match.pas A game, place "match" a number game.

match.cmp Compare file for match automated testing.

match.inp Input file for match automated testing.

pascals.pas Niklaus Wirth’s Pascal-s subset interpreter..

pascals.cmp Compare file for pascals automated testing.

pascals.inp Input file for pascals automated testing.

roman.pas A slightly more complex test program, prints roman numerals. From Niklaus Wirth's "User Manual and Report".

roman.inp Input file for roman automated testing.

roman.cmp Compare file for roman automated testing.

startrek.pas The startrek game.

startrek.inp Startrek input file.

startrek.cmp Startrek compare file.

## Directory: standard\_tests

iso7185pat.cmp Contains the output from the PAT file with the IP Pascal compiled P6 executables above.

iso7185pat.inp The input file for the Pascal acceptance test.

iso7185pat.pas The Pascal Acceptance Test. This is a single Pascal source that tests how well a given Pascal implementation obeys ISO 7185 Pascal. It can be used on P6 or any other Pascal implementation.

iso7185pats.cmp Contains the output from the PAT file resulting from the cpints run. This is slightly different than the normal run.

iso7185prt.bat Compiles the Pascal rejection tests.

iso7185prtXXXX.cmp Comparision file for pascal rejection test. XXXX is a four digit number. See the rejection tests text for information.

Iso7185prtXXXX.inp Contains the input file for the Pascal rejection test XXXX.

Iso7185prtXXXX.pas Contains the source file for the Pascal rejection test XXXX.

# The intermediate language

## Format of intermediate

The intermediate has the format:

<main code>

<start code>

<further prd input>

The main section contains all of the generated code, except for a startup section:

mst 0

cup 0 l 3

stp

The main section is assembled past the startup code, then the assembly location is restarted and the startup section is assembled, placed under the main code.

After both the main code and start code sections, the contents of the prd file are not read. The interpreted code can keep reading from the prd file at this point. This means that input for the prd file as used by the interpreted program can be concatenated to the intermediate file. This feature is used for the self compile and run.

For each line in the intermediate, the first character indicates:

i Indicates a comment, the rest of the line is discarded.

l A label. Used to establish jump locations in the code.

q Marks the end of the main code or startup code section.

(blank) An intermediate instruction.

: A source line marker.

Comments

A comment appears as:

i<any text>

Comments are used to generate any descriptive text in the intermediate. They are also used to output a marker every 10 intermediate instructions, of where the “logical program count” or index of instructions, is currently located in the intermediate with a marker of the form:

i n

Where n is the logical program count.

### Label

Label lines are of the format:

l n[= val]

The label number gives the logical label being defined. If “=” follows, the instruction address of the label appears, otherwise it is set to the current instruction being processed (the pc).

The logical label number is a value from 0 to n. The value can be anything, and it is used both to define parameters as well as addresses. When used in an instruction, the assembler is capable of processing forward references to labels not yet defined in the assembly.

### Source line marker

A source line marker gives the line number currently being parsed in the compiler when the intermediate is generated. It is of the form:

:n

Where n is the source line number. This information can be used to show where errors occurred in the source.

## Intermediate instruction set

What follows is a complete listing of all instructions, parameters, and numeric equivalences in the interpreter instruction set. The following instruction endings are common and indicate the type of the operation:

i Integer (4 bytes)

r Real (8 bytes)

s Set (32 bytes)

b Boolean (1 byte)

a Address (4 bytes)

m Memory (or memory block)

c Character (1 byte)

Note that all boolean and character values are extended to 4 byte integers when loaded to the stack. Reals and sets are loaded as whole entities, 8 and 32 bytes respectively.

Note that there will often not be a type indicator if there is only one type operated on by the instruction.

The operation codes are from 0 to 255 or $00 to $ff, fitting in one unsigned byte. The format of the opcodes and operands is:

Op [p] [q [q1]]

The p parameter, if it is exists, is always an unsigned byte. The q parameter, if it exists, is either one or 4 bytes long. If q1 exists, q is always 4 bytes long. q1 is always 4 bytes long. These will be refered to below as q8 and q32 (8 bit and 32 bits). The endian nature of 32 bit words will depend on the compiler used for P6.

Each instruction is listed first in the form it appears in the intermediate assembly form. The second is the format the opcode will take in interpreter store.

This list of instructions is in alphabetical order.

Instruction Opcode Stack in Stack out

abi 40 integer integer

abr 41 real real

Find absolute value of integer or real on top of stack.

Instruction Opcode Stack in Stack out

adi 28 integer integer integer

adr 29 real real real

Add the top two stack integer or real values and leave result on stack.

Instruction Opcode Stack in Stack out

and 43 boolean boolean boolean

Find logical ‘and’ of the top two Booleans on stack, and replace with result.

Instruction Opcode Stack in Stack out

chka low high 95 q32 address address

chkb low high 98 q32 boolean boolean

chkc low high 99 q32 char char

chki low high 26 q32 integer integer

chks low high 97 q32

Bounds check. The q parameter contains the address of a pair of low and high bounds values, low followed by high, each one integer in size. The bounds pair is placed in the constant area during assembly, and the address placed into the instruction.

The value on top of the stack is verified to lie in the range low..high. If not, a runtime error results. The value is left on the stack.

If the instruction was chka, the low value will contain the code:

0 Pointer is not being dereferenced

1 Pointer is being dereferenced

Pointer values are considered valid if between 0..maxaddr, where maxaddr is the maximum address in the interpreter. If the dereference code is present, it indicates a check if the pointer is nil. This instruction is only applied to dynamic variable addresses, and thus other checks are possible.

Set values are checked to see if any elements outside the elements from low..high are set. If so, a runtime error results.

Instruction Opcode Stack in Stack out

chr 60 integer character

Find character from integer. Finds the character value of an integer on stack top. At present, this is a no-op.

Instruction Opcode Stack in Stack out

cip p 113 mp address

Call indirect procedure/function. The top of stack has the address of a mp/address pair pushed by lpa. The dl of the current mark is replaced by the mp, and the address replaces the current pc. The mp/ad address is removed from stack. The size of the parameters appears as p, which shows where to find the mark.

Instruction Opcode Stack in Stack out

cke 188 tagcst boolean

Terminate active variant check, started by cks. Expects the “running boolean” value on stack top, followed by the current value of the tagfield. If the Boolean value is false, meaning that none of the constants matched the tagfield, a runtime error results.

Both the Boolean value and the tagfield value are removed from the stack.

Instruction Opcode Stack in Stack out

ckla low high 190 address address

Bounds check. This instruction is identical to the instruction chka, but indicates a pointer to a tagged record. At present this requires no special action over the normal processing of chka.

ckla an chka check if the pointer is dereferencing an allocation that has been freed. In the current implementation this check works on both tag listed and normal dynamic variables. Because this is dependent on the way the allocator is structured, ckla exists in case tag list pointers need special handling.

Instruction Opcode Stack in Stack out

cks 187 tagval tagval boolean

Start active variant check. The current value of the tag is on stack top. Pushes a false binary value onto the stack. This value is ‘or’ed with all of the following variant checks.

Instruction Opcode Stack in Stack out

ckvb val 179 q32 tagval Boolean tagval boolean

ckvc val 180 q32

ckvi val 175 q32

Checks a tagfield for active variants. The “running Boolean” is expected at stack top, followed by the current tagfield value. The tagfield is compared to the constant val and the Boolean equality ‘or’ed into the running Boolean. Both are left on stack.

Instruction Opcode Stack in Stack out

csp rout 15 q8 Per call Per call

Call system routine. The routine number rout indicates the routine to call, from 0 to 255 (or $00 to $ff). Note that the main difference between system and user procedures is that system calls have no framing information.

Instruction Opcode Stack in Stack out

cta off lvl 191 q32 q32 tagaddr newtagval tagaddr newtagval

Check tag assignment for dynamically allocated record. Expects the value to assign to the tagfield on stack top, followed by the address of the tagfield as second on stack. Dynamic records containing tagfields are allocated by a special procedure that creates a tagfield constant list with the constants that were used to allocate the record with variants. The cta instruction looks at the list to see if the given tagfield appears in that list, indicating that the tagfield is fixed and cannot be changed. If such a list entry exists and does not match the tagfield assignment, a runtime error results. The instruction contains the offset from the start of the record to the tagfield, and at what nesting level the tagfield exists. If uses the tagfield offset to find the start of record and look for the list, and the nesting level to find the exact tag constant, if it exists. Leaves both values on stack.

Instruction Opcode Stack in Stack out

cup p l addr 12 p8 q32

Call user procedure. The instruction contains the number of bytes in the parameter section, and the address of the procedure to call. The mp or mark pointer is set to the location of the stack mark record established by the mst instruction, then the return address is placed into the mark record. The specified location is then jumped to.

The assembly code represents the address as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

decb cnt 103 q32 boolean boolean

decc cnt 104 q32 character character

deci cnt 57 q32 integer integer

The unsigned cnt parameter is subtracted from the value at stack top.

Instruction Opcode Stack in Stack out

dif 45 set set set

Find set difference, or s1-s2. The members of the set on stack top are removed from the set that is second on the stack, and the top set on stack is removed.

Instruction Opcode Stack in Stack out

dmp cnt 117 q32 value

The unsigned cnt parameter is subtracted from the current stack pointer. The effect is to “dump” or remove the topmost cnt data from the stack.

Instruction Opcode Stack in Stack out

dupa 182 address address address

dupb 185 boolean boolean boolean

dupc 186 character character character

dupi 181 integer integer integer

dupr 183 real real real

dups 184 set set set

Duplicate stack top. The top value on the stack is copied to a new stack top value according to type.

Instruction Opcode Stack in Stack out

dvi 53 integer integer integer

dvr 54 real real real

Divide. The value second on stack is divided by the value first on stack. The division is done in integer or real according to type.

Instruction Opcode Stack in Stack out

ente l size 173q32

Sets the maximum extent of stack use. The size constant in the instruction gives a compiler calculated size that is the maximum extent of stack that will be used by this routine. If that overlaps the bottom of the heap, then a runtime error results. The ep or extreme stack pointer is updated, and also stored in the stack mark record for use in the ipj instruction to jump between procedures.

The assembly code represents the size as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

ents l size 13 q32

Enter routine and allocate local variable space. The size field in the instruction contains the size of data for the routine being entered, including stack mark, parameters, and locals. The mp or mark pointer is offset by size, the area between the sp or stack pointer and the new allocation is cleared to zero and the sp set after that. The bottom of the stack field in the stack mark record is set from that, which allows the ipj instruction to find the proper setting of the stack for interprocedural jumps.

Checks if the resulting locals allocation would overrun the heap, and thus require more memory that is available. A runtime error results if not.

The assembly code represents the size as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

equa 17 address address boolean

equb 139 boolean Boolean boolean

equc 141 character character boolean

equi 137 integer integer boolean

equm size 142 q32 address address boolean

equr 138 real real boolean

equs 140 set set boolean

Find equal. The top of stack and second on stack values are compared, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

fjp addr 24 q32 boolean

Jump false. Expects a Boolean value atop the stack. If the value is false, or zero, execution continues at the instruction constant addr. Removes the boolean from stack.

Instruction Opcode Stack in Stack out

flo 34 integer real real real

flt 33 integer real

Convert stack integer to floating point. flt converts the top of stack to floating point from integer. flo converts the second on stack to floating point, but it assumes that the top of stack is real as well.

Instruction Opcode Stack in Stack out

geqb 151 boolean boolean boolean

geqc 153 character character boolean

geqi 149 integer integer boolean

geqm size 154 q32 address address boolean

geqr 150 real real boolean

geqs 152 set set boolean

Find greater than or equal. The top of stack and second on stack values are compared for second on stack greater than or equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

grtb 157 boolean boolean boolean

grtc 159 character character boolean

grti 155 integer integer boolean

grtm size 160 q32 address address boolean

grtr 156 real real boolean

grts 158 set set boolean

Find greater than. The top of stack and second on stack values are compared for second on stack greater than the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

inca cnt 90 q32 address address

incb cnt 93 q32 boolean boolean

incc cnt 94 q32 character character

inci cnt 10 q32 integer integer

Increment top of stack. The unsigned constant cnt is added to the value on stack top. Note that type a is not subject to arithmetic checks, but the other types are.

Instruction Opcode Stack in Stack out

inda off 85 q32 address address

indb off 88 q32 address boolean

indc off 89 q32 address character

indi off 9 q32 address integer

indr off 86 q32 address real

inds off 87 q32 address set

Load indirect to stack. Expects the address of an operand in memory on the stack top. The constant off is added to the address, then the operand fetched from memory and that replaces the address on stack top.

Instruction Opcode Stack in Stack out

inn 48 value set boolean

Set inclusion. Expects a set on stack top, and the value of a set element below that. Tests for inclusion of the value in the set, then replaces both of them with the Boolean result of that test.

Instruction Opcode Stack in Stack out

int 46 set set set

Set intersection. Finds the intersection of the two sets on the stack and replaces them both with the resulting set.

Instruction Opcode Stack in Stack out

inv 189 address2

Invalidate address. Expects an address on stack, then flags that location as undefined if undefined checking is enabled. This instruction is used to return variables to the undefined state. Removes the address from stack.

Instruction Opcode Stack in Stack out

ior 44 boolean boolean boolean

Boolean inclusive ‘or ‘. Expects to find two Boolean values on stack. Replaces them with the inclusive ‘or’ of the values.

Instruction Opcode Stack in Stack out

ipj p l addr 112 p8 q32

Interprocedure jump. The instruction contains the relative frame number p, and the address to jump to within the target procedure addr. The frames above the target procedure frame are discarded, and the target frame is loaded. Execution then proceeds at the given address addr.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

ivt off size 192 q32 q32 tagaddr newtagval tagaddr newtagval

Invalidate tagged variant. Expects a new setting for a tagfield at stack top, and the address of the tagfield below that. If the tagfield was defined, and the new tag value is different than the old tag value, the variant area controlled by the tagfield is set as undefined. The off instruction field constains the offset from the tagfield to the base of the variant. The size instruction field contains the size of the variant. The offset is applied to the address of the tagfield, and the entire variant is set undefined. Note that the size of the variant is the maximum size of all possible variants.

The stack is unchanged during this operation.

Instruction Opcode Stack in Stack out

ixa size 16 q32 address index address

Scale array access. Expects an index value at stack top, and the address of an array below that. The size instruction field contains the size of the base element of the array. The index is multiplied by the size, then added to the address of the base of the array, then that element address replaces both the index and the base address on stack.

Note that this operation does not remove the array index offset, ie., array [1..10] does not have 1 removed here.

Instruction Opcode Stack in Stack out

lao off 5 q32 address

Load address with offset. Loads the address in the globals area with offset off. This instruction is used to index globals. The address is placed on stack top.

Instruction Opcode Stack in Stack out

lca len ‘str’ 56 q32 address

Load string address. P6 accepts any length of string between quotes, but stores the string as the given length. The string is padded as required with spaces. Accepts a “quote image” in the string.The string is placed in the constants area, and the instruction gets the address of that. Loads the string constant address off to the top of the stack.

Instruction Opcode Stack in Stack out

lda p addr 4 p8 q32 address

Load local address. Expects the relative frame number of the target procedure in p, and the offset address of the local in addr. The frame is found and the address calculated as an offset from the locals there and placed on the stack.

Instruction Opcode Stack in Stack out

ldcs ( set ) 7 q32 set

Load constant set. The address in the instruction gives the address of the set, which is loaded to stack top. This is used to load sets from the constant area. The set is specified as a series of values in the set between parenthsis. This set is loaded into the constant area on assembly, and the instruction contains the address of that.

Instruction Opcode Stack in Stack out

ldcb bool 126 q8 boolean

ldcc char 127 q8 character

ldci int 123 q32 integer

Load constant. Loads a constant from within the instruction according to the type, Boolean, character or integer.

Instruction Opcode Stack in Stack out

ldcn 125 nilcst

Load nil value. Loads the value of nil to the stack top.

Instruction Opcode Stack in Stack out

ldcr real 124 q32 real

Load constant real. The address in the instruction gives the address of the real, which is loaded to stack top. The real given is loaded into the constants area on assembly, and the instruction contains the address of that.

Instruction Opcode Stack in Stack out

ldoa off 65 q32 address

ldob off 68 q32 boolean

ldoc off 69 a32 character

ldoi off 1 q32 integer

ldor off 66 q32 real

ldos off 67 q32 set

Load from offset. Loads a global value from the globals area. The off constant gives the offset of the variable in the stack area. The value is loaded to the top of stack according to type.

Instruction Opcode Stack in Stack out

leqb 163 boolean boolean boolean

leqc 165 character character boolean

leqi 161 integer integer boolean

leqm size 166 q32 address address boolean

leqr 162 real real boolean

leqs 164 set set boolean

Find less than or equal. The top of stack and second on stack values are compared for second on stack less than or equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

lesb 169 boolean boolean boolean

lesc 171 character character boolean

lesi 167 integer integer boolean

lesm size 172 q32 address address boolean

lesr 168 real real boolean

less 170 set set boolean

Find less than. The top of stack and second on stack values are compared for second on stack less than the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

lip p addr 120 p8 q32 mp pfaddr

load procedure function address. Expects the relative mark count p, and the address of an existing mark/function address pair in the instruction. Loads a mark/address pair for a procedure or function parameter onto the stack from a previously calculated pair in memory. Used to pass a procedure or function parameter that was passed to the current function to another procedure or function. See the cip instruction for further information.

Instruction Opcode Stack in Stack out

loda p off 105 Address

lodb p off 108 boolean

lodc p off 109 character

lodi p off 0 integer

lodr p off 106 real

lods p off 107 set

Load local value. Expects the display offset p and the offset address in the local procedure frame off in the instruction. The value is loaded according to type.

Instruction Opcode Stack in Stack out

lpa p l addr 114 p8 q32 mp pfaddr

Load procedure address. The current mark pointer is loaded onto the stack, followed by the target procedure or function address. This puts enough information on the stack to call it with the callers environment.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

mod 49 integer integer integer

Find modulo. Finds the modulo of the second on stack by the top of stack. The result replaces both operands. This is always an integer operator.

Instruction Opcode Stack in Stack out

mov size 55 destaddr srcaddr

Move bytes. The instruction contains the number of bytes to move. The top of stack contains the source address, and the second on stack contains the destination address. The specified number of bytes are moved. Both addresses are removed from the stack.

Instruction Opcode Stack in Stack out

mpi 51 integer integer integer

mpr 52 real real real

Multiply. The first and second on stack are multiplied together, and the result replaces them both.

Instruction Opcode Stack in Stack out

mrkl line 174

Mark source line. This instruction exists only within the interpreter, and is not specified in the intermediate. The instruction parameter line contains the number of the source line that was read in at this point in the intermediate generation. The interpreter can use this to provide various source level debug services.

Instruction Opcode Stack in Stack out

mst p 11 p8

Mark stack. This instruction is used before parameters are loaded in expectation of a user procedure call. The p parameter in the instruction contains the level of the calling procedure minus the level of the called procedure and is used to establish the static links in the mark. A stack mark record is placed onto the stack an the sl or static link is placed, the dl or dynamic link, and the ep or extended stack pointer.

Instruction Opcode Stack in Stack out

neqa 18 address address boolean

neqb 145 boolean Boolean boolean

neqc 147 character character boolean

neqi 143 integer integer boolean

neqm size 148 address address boolean

neqr 144 real real boolean

neqs 146 set set boolean

Find not equal. The top of stack and second on stack values are compared for second on stack not equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

ngi 36 integer integer

ngr 37 real real

Negate. The operand atop the stack is negated according to type.

Instruction Opcode Stack in Stack out

not 42 boolean boolean

Logical ‘not’. The Boolean value atop the stack is inverted.

Instruction Opcode Stack in Stack out

odd 50 integer boolean

Find odd/even. The lowest bit of the integer on stack is masked to find the even/odd status of the integer as a Boolean value.

Instruction Opcode Stack in Stack out

ordb 134 boolean integer

ordc 136 character integer

ordi 59 integer integer

Find ordinal value of Boolean, character or integer. This is a no-op, because P6 always converts values to 32 bit integers on load to stack.

Instruction Opcode Stack in Stack out

pck sizep sizeu 63 q32 q32 uparr inx parr

Convert unpacked array to packed array. Because P6 does not support packing, this is effectively a copy operation. The instruction contains the size of the packed array sizep and the size of the unpacked array sizeu. The stack contains the address of the packed array at top, the starting index of the unpacked array under that, and the unpacked array address as third on stack. The number of elements in the packed array are moved from the unpacked array at the starting index to the packed array. All parameters are removed from the stack.

Instruction Opcode Stack in Stack out

reta 132 address

retb 131 boolean

retc 130 character

reti 128 integer

retp 14

retr 129 real

Return from procedure or function with result. Returns from the current procedure or function. The pc, sp, ep and mp pointers are restored from the saved data in the active stack mark record. For retp or return from procedure, this is all that is done. For the others, a return value is processed according to type. The value of the result is loaded from its place in the stack mark record, and expanded to 32 bits if Boolean or character.

Instruction Opcode Stack in Stack out

rgs 110 low high set

Build range set on stack. Expects a high value on stack top, and a low value below that. A set is constructed that has all of the members from the low value to the high value in it, then that set is placed as stack top.

The range builder instruction, along with sgs or build singleton set, is used to construct complex sets by creating sets of each individual ranges of values, then adding the resulting sets together on stack.

Instruction Opcode Stack in Stack out

rnd 62 real integer

Round real value to integer. Converts the real on top of the stack to integer by rounding.

Instruction Opcode Stack in Stack out

sbi 30 integer integer integer

sbr 31 real real real

Subtract. Subtracts the top of stack value from the second on stack value according to type.

Instruction Opcode Stack in Stack out

sgs 32 integer set

Construct singleton set. Expects a value on stack. Constructs a set with that value as the single member, then that set replaces the value on stack.

The singleton set instruction, along with rgs or range set builder, is used to construct complex sets by creating sets with individual ranges of values, then adding the resulting sets together on stack.

Instruction Opcode Stack in Stack out

sqi 38 integer integer

sqr 39 real real

Find square of value. Expects a value on stack top, and finds the square of that value according to type. This replaces the value on stack top.

Instruction Opcode Stack in Stack out

sroa off 75 address

srob off 78 boolean

sroc off 79 character

sroi off 3 integer

sror off 76 real

sros off 77 set

Store global value. The instruction contains the offset of a variable in the stack bottom, where the globals for the program exist. The variable is stored from the stack according to type, and the value removed from the stack.

Instruction Opcode Stack in Stack out

stoa 80 address address

stob 83 address boolean

stoc 84 address character

stoi 6 address integer

stor 81 address real

stos 82 address set

Store value to address. Expects a value according to instruction type atop the stack, and an address below that. The value is stored to the address, and both are removed from the stack.

Instruction Opcode Stack in Stack out

stp 58

Stop. Executing this instruction causes the interpreter to exit.

Instruction Opcode Stack in Stack out

stra p off 70 address

strb p off 73 boolean

strc p off 74 character

stri p off 2 integer

strr p off 71 real

strs p off 72 set

Store local. Instruction parameter p contains the relative count of the procedure frame to access. Instruction parameter off contains the offset address within the frame. The value at stack top is stored to the local variable according to type, and the value removed from the stack.

Instruction Opcode Stack in Stack out

swp size 118 value address address value

Swap first and second stack operands. The instruction contains the size of the second value on stack. The top value is assumed to be 32 bit. The top and second values are swapped. This instruction is used primarily to chain writes together to the same file access pointer.

Instruction Opcode Stack in Stack out

tjp l addr 119 boolean

Jump true. Expects a Boolean value on stack top. If the Boolean is true, the jump is taken to address addr. The Boolean is removed from the stack.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

trc 35 real integer

Truncate to integer. The real on stack top is converted to integer by truncating the fractional part. The integer replaces the real on stack top.

Instruction Opcode Stack in Stack out

ujc 61 q32

Output case error. This instruction is used in case statement tables to output case value errors. See xjp for the case table format. It is designed to be the same length as ujp, which is used to build the case table. The 32 bit address in the struction is a dummy, and contains zero. Always outputs a bad case select error. Inserted into the table where the value for the case would be invalid.

Instruction Opcode Stack in Stack out

ujp l addr 23

Unconditional jump. The instruction contained address addr is jumped to.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

uni 47 set set set

Find set union. Expects two sets on stack. The union of the sets is found, and that replaces them both.

Instruction Opcode Stack in Stack out

upk sizep sizeu 64 q32 q32 packarr upackarr sindex

Unpack a packed array. P6 does not implement packing, so this instruction is implemented as a copy operation. Expects the size of the packed array sizep and the size of the unpacked array sizeu in the instruction. The stack contains the starting index of the unpacked array at top, followed by the address of the unpacked array, and the address of the packed array as third on stack. The entire contents of the packed array are transferred into the unpacked array, which can be larger than the packed array.

Instruction Opcode Stack in Stack out

xjp l addr 25 q32 index

Table jump. Expects an address of a jump table within the instruction, and a jump table index on the stack. The jump table constists of a series of jump instructions using the ujp instruction. The index is multiplied by the length of the ujp instruction, which is 5 bytes long, and the pc set to that address. The effect is to jump via a table of entries from 0 to n. The index is removed from stack.

This instruction is used to implement the case statement.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

## System calls

The P6 interpreter partitions the execution work done in the stack machine into directly executed instructions and system calls. The system call instructions consist of I/O related calls and dynamic storage allocation. These are normally things that are handled by the runtime support system.

The difference between a function executed in an instruction and a function executed in a system call is minimal. A system call uses parameters stacked onto the interpreter operand stack in reverse just as the intermediate does.

The difference between a function executed in a user based procedure or function and a function executed in a system call is that the system call requires no framing or locals.

System calls are executed by number via the csp instruction.

The following system call descriptions show the intermediate menmomic of the call, the instruction number used as a parameter to the csp instruction, and the operands as they exist on the stack before the call. The stack operands are listed deepest first at left, to the topmost operand at the right. This is the logical call order of the routine parameters.

Each system call can be a procedure or a function. If it is a function, it leaves its result on the stack.

System Call Number Stack in Stack out

atn 19 real real

Find arctangent. Expects a real on stack. Finds the arctangent, and that replaces the stack top.

System Call Number Stack in Stack out

cos 15 real real

Find cosine. Expects a real on stack. Finds the cosine, and that replaces the stack top.

System Call Number Stack in Stack out

dsl 40 addr tagcst.. tagcnt size

Dispose of dynamic tagged record. This is a special form of dispose for a dynamically allocated record with tagged variants. The top of the stack contains the size of the record, after any fixed tag constants specified are taken into account. Below that, the number of tagfield constants that were specified exists. Below that, a list of all the constants that were used to specify the allocation. This list is from leftmost in sourcecode order and deepest in the stack to rightmost in source and topmost in stack. Following that, the address of the dynamic record.

The allocation of the record, done by the system call nwl, contains a matching list of tagfield constants allocated “behind” the allocated pointer, with the number of tag constants last in the list so that it can be found just below the pointer. These lists represent the tagfield list used to allocate the record, in the dynamic itself, and the tagfield list used to dispose of it, on stack. These lists are compared for both number and value, and a runtime error results if not equal. The total allocation, record plus taglist, is then disposed of.

System Call Number Stack in Stack out

dsp 26 addr size

Dispose of dynamic variable. The top of stack contains the variable size, and the address under that is the address of the dynamic record. The variable is deallocated. Both operands are removed from stack.

System Call Number Stack in Stack out

efb 42 fileaddr boolean

Find eof of binary file. Expects the address of a file variable on stack. The file is tested for eof() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

eln 7 fileaddr boolean

Test for eoln of text file. Expects the address of a file variable on stack. The file is tested for eoln() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

eof 41 fileaddr boolean

Find eof of text file. Expects the address of a file variable on stack. The file is tested for eof() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

exp 16 real real

Find exponential. Expects a real on stack. Finds the exponential, and that replaces the stack top.

System Call Number Stack in Stack out

fbv 43 fileaddr fileaddr

File buffer validate text. Expects the address of a file variable on stack. Ensures the file buffer variable is loaded. If the file is a read file, or the file is in read mode, the file buffer variable is loaded by reading the file. The file address pointer is left on stack. This call is used to insure the file buffer contains data from the file, if it exists, when the file buffer variable is referenced. It is part of the “lazy I/O” file scheme.

System Call Number Stack in Stack out

fvb 44 fileaddr size fileaddr

File buffer validate binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. Ensures the file buffer variable is loaded. If the file is a read file, or the file is in read mode, the file buffer variable is loaded by reading the file. The element length tells the call how many bytes to read. The file address pointer is left on stack, but the size is purged. This call is used to insure the file buffer contains data from the file, if it exists, when the file buffer variable is referenced. It is part of the “lazy I/O” file scheme.

System Call Number Stack in Stack out

gbf 35 fileaddr size

Get file buffer binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. If the file buffer variable is indicated as full, then it is marked empty, thus discarding the data there. If it is full, new data is read over the file buffer variable. Both the file address pointer and the size are discarded from stack. This operation is part of the “lazy I/O scheme”.

System Call Number Stack in Stack out

get 0 fileaddr

Get file buffer text. Expects the address of a file variable on stack. Reads the next element of the file into the file variable buffer. The file pointer is discarded.

System Call Number Stack in Stack out

log 17 real real

Find logarithm. Expects a real on stack. Finds the logarithm, and that replaces the stack top.

System Call Number Stack in Stack out

new 4 addr size

Allocate dynamic variable. Expects the size of variable in bytes to allocate on stack top, followed by the address of the pointer variable. Space with the given size is allocated, and the address placed into the pointer variable. Both operands are removed from stack.

System Call Number Stack in Stack out

nwl 39 addr tagcst.. tagcnt size

Allocate dynamic tagged record. This is a special form of new() for a dynamically allocated record with tagged variants. The top of the stack contains the size of the record, after any fixed tag constants specified are taken into account. Below that, the number of tagfield constants that were specified exists. Below that, a list of all the constants that were used to specify the allocation. This list is from leftmost in sourcecode order and deepest in the stack to rightmost in source and topmost in stack. Following that, the address of the pointer variable.

The space required for the tagfield constant list and length is added to the total space required, and that space is allocated. The tagfield constant list, with the following length, is copied to the start of the allocated space, and the pointer variable is set just after that. Thus the tagfield constant list exists “behind” the pointer or in “negative space”, so that it can be found and referenced for various purposes. These are:

1. To check that the tagfield constant list is equal between new() and dispose() operations.
2. To check if a tagfield used to set the total space allocated for the variable is assigned with a different value than originally used in the new() call.

The size, tagfield list and length, and the address of the pointer variable are discarded.

System Call Number Stack in Stack out

pag 21 fileaddr

Page text file. Expects the address of a file variable on stack. A page request is sent to the file, and the file address is discarded.

System Call Number Stack in Stack out

pbf 36 fileaddr size

Put file buffer binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. If the file buffer variable is empty, a runtime error results. Otherwise, the contents of the file buffer variable are written out to the file. The size and file address are discarded.

System Call Number Stack in Stack out

put 1 fileaddr

Put file buffer text. Expects the address of a file variable on stack. If the file buffer is empty, a runtime error results. Otherwise, writes the contents of the file buffer variable to the file. The file address is discarded.

System Call Number Stack in Stack out

rbf 32 fileaddr varaddr len fileaddr

Read binary file. Expects a file variable address on stack, the address to place read data above that, and the length of the base file element on stack top. If the buffer for the file is has data, then the data is read from that, otherwise the data is read from the file. The number of bytes in the element length are read.

Purges the target variable address and the element length from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rcb 38 fileaddr vaddr min max fileaddr

Read character from text file with range check. Expects the file variable address on stack, the variable address to read to above that, then the minimum value min and the maximum value max at stack top. Reads a single character from the file, verifies that it lies in the specified range, and places it to the variable. If the character value lies out of range, a runtime error results.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdc 13 fileaddr varaddr fileaddr

Read character from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single character from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdi 11 fileaddr varaddr fileaddr

Read integer from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single integer from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdr 12 fileaddr varaddr fileaddr

Read real from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single real from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rib 37 fileaddr vaddr min max fileaddr

Read an integer from text file with range check. Expects the file variable address on stack, the variable address to read to above that, then the minimum value min and the maximum value max at stack top. Reads a single integer from the file, verifies that it lies in the specified range, and places it to the variable. If the character value lies out of range, a runtime error results.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rln 3 fileaddr fileaddr

Read next line from text file. Expects the file variable address on stack. The text file is read until an eoln() or eof() is encountered.

Leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rsb 33 fileaddr

Reset file binary. Expects the file variable address on stack. The binary file is reset(). The file variable address is purged.

System Call Number Stack in Stack out

rsf 22 fileaddr

Reset file text. Expects the file variable address on stack. The text file is reset(). The file variable address is purged.

System Call Number Stack in Stack out

rwb 34 fileaddr

Rewrite file binary. Expects the file variable address on stack. The binary file is rewritten. The file variable address is purged.

System Call Number Stack in Stack out

rwf 23 fileaddr

Rewrite file text. Expects the file variable address on stack. The text file is rewritten. The file variable address is purged.

System Call Number Stack in Stack out

sin 14 real real

Find sine. Expects a real on stack. Finds the sine, and that replaces the stack top.

System Call Number Stack in Stack out

sqt 18 real real

Find square root. Expects a real on stack. Finds the square root, and that replaces the stack top.

System Call Number Stack in Stack out

wbb 31 fileaddr boolean fileaddr

Write Boolean to binary file. Expects the file variable address on stack, and the Boolean to write to above that. Writes a single boolean to the file from the variable.

Purges the boolean from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbc 30 fileaddr char fileaddr

Write character to binary file. Expects the file variable address on stack, and the Boolean to write to above that. Writes a single character to the file from the variable.

Purges the character from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbf 27 fileaddr varaddr size fileaddr

Write binary variable to binary file. Expects the file variable address on stack, the address of the variable to write above that, and the size in bytes of the variable at stack top. Writes all the bytes of the variable to the file single boolean to the file from the variable.

Purges the size and the variable address from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbi 28 fileaddr integer fileaddr

Write integer to binary file. Expects the file variable address on stack, and an integer to write to above that. Writes a single integer to the file from the variable.

Purges the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbr 29 fileaddr real fileaddr

Write a real to binary file. Expects the file variable address on stack, and a real to write to above that. Writes a single real to the file from the variable.

Purges the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wln 5 fileaddr fileaddr

Write next line to text file. Expects the file variable address on stack. A new line is written to the text file.

Leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrb 24 fileaddr boolean width fileaddr

Write boolean to text file. Expects the file variable address on stack, a boolean to write to above that., and a field width at stack top. Writes the Boolean in the given width.

Purges the width and the boolean from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrc 10 fileaddr char width fileaddr

Write character to text file. Expects the file variable address on stack, a character to write to above that., and a field width at stack top. Writes the character in the given width.

Purges the width and the character from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrf 25 fileaddr real width frac fileaddr

Write real to text file in fixed point notation. Expects the file variable address on stack, a real to write to above that., a field width above that, and a fraction at stack top. Writes the real in the given width.

Purges the fraction, width and the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wri 8 fileaddr integer width fileaddr

Write integer to text file. Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrr 9 fileaddr real width fileaddr

Write real to text file in floating point notation. Expects the file variable address on stack, a real to write to above that., and a field width at stack top. Writes the real in the given width.

Purges the width and the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrs 6 fileaddr saddr width len fileaddr

Write string to text file. Expects the file variable address on stack, the address of the string to write to above that., the field width above that, and the length of the string at stack top. Writes the string in the given width.

Purges the length, width and the string address from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

# Testing P6

In the original implementation of Pascal and the Pascal-P porting kit, the implementation of tests on the system were largely undefined. Today, most programmers realize that any program exists as a collection of the code, documentation, and finally the tests for the program to prove it correct. If any one of these elements is missing, much as a three legged stool, the program will fall over. In reality, an “undocumented” program that is popular gets documented by its users or by third parties, such as independent book writers. Tests can be carried out ad-hoc, or essentially fall to the users (their happiness with this situation being quite another matter!).

Thus, as important to P6 as its code or documents are its tests.This was widely recognized with original Pascal, and an extensive series of tests were created with the advent of the ISO 7185 standard, as documented in “Pascal compiler validation” [Brian Wichmann & Z. J. Chechanowicz]. This was an excellent series of tests that showed close relationship to the standard. Unfortunately, like the “model compiler”, the rights to this test series, which was initially openly distributed, were closely held, and the project is essentially dead today.

Accordingly, I have created a new series of tests for P6, and now that is distributed with P6 itself, and continues to be improved. The PAT or Pascal Acceptance Test is completely automated. Further, because the original ISO 7185 validation tests were distributed in the “Pascal User’s Group” free of restrictions, I have used them to check on the completeness of the PAT. Thus, even if the original ISO 7185 validation tests cannot legally be distributed, the PAT is tracable back to these tests and is a reliable substitute for them.

## Running tests

### testprog

The main test script for testing is:

testprog <Pascal source file>

testprog.bat <Pascal source file>

testprog is a “one stop” test resource for most programs. It expects the following files to exist under the given primary filename:

program.pas The Pascal source file.

program.inp The input file for the running program.

program.cmp The reference file for the expected output.

testprog compiles and runs the target program, and checks its output against the reference file. Several files are produced during the process:

program.P6 Contains the intermediate code for the program as compiled by pcom.

Program.err Contains the output from pcom. This includes the status line indicating the number of errors. It also typically contains a listing of the program as compiled, which contains line numbers and other information. Note that if the pcom compile run produces errors, testprog will stop.

Program.lst Contains the output from the program when run. This is all output to the standard “ output” file.

Program.out This is all of the output to the “prr” special file. This is only used by P6 for special purposes, such as to output the intermediate code, and most programs will be empty.

Program.dif This is the output of the diff command between program.lst and program.cmp. It should be empy if the program produced the output expected.

Not all of the files for testprog need to have contents. For example, a program that does not do input does not need to have a program.inp file. An example of this would be the “hello” program. However, testprog expects all of the files to exist.

To determine if the test ran correctly, the output of first the program.err file should be checked for zero errors, then the resulting program.dif file is checked for zero length. All of these results are announced during the test:

C:\projects\PASCAL\P6\_ext>testprog sample\_programs\roman

Compile and run sample\_programs\roman

Compile fails, examine the sample\_programs\roman.err file

For a program that has a compile error.

C:\projects\PASCAL\P6\_ext>testprog sample\_programs\roman

Compile and run sample\_programs\roman

03/18/2012 10:44 AM 76 roman.dif

For a program that does not match it’s expected output.

If a large series of test programs are compiled and run with testprog, it may be more efficient to examine the output files to determine the result, first the program.err file, then the program.dif file. The rejection test is a good example of such a large series of files.

### Other tests

Not all test programs work well with the testprog script. Examples are “pascals”, and the self compile. For these programs a special test script is provided.

### Regression test

All of the “positive” tests are wrapped up in a single script, “regress.bat”. This script runs all of the positive tests in order of successive complexity. Whenever a change is made to P6, the regression test is run to verify that the resulting compiler is still valid.

## Test types

The Pascal-P6 tests are divided into two major catagories, referred here as “positive” and “negative” tests. The positive tests are designed to test what the compiler should accept as a valid program. The negative tests are designed to test what the compiler should reject as invalid programs.

Thus, they are termed here as the “Pascal Acceptance” test, and the “Pascal rejection test”. The two test types are fundamentally different. For example, all of the acceptance tests can appear in a single program, since the entire program should compile and run as a valid program. However, the Pascal rejection test cannot practically be represented as a single program because it would (or should) consist of a repeated series of errors. The compiler could conceivably recover from the each error sufficiently to encounter and properly flag the next error, but this would not be reliable, and would vary from compiler to compiler. Further, such a test would test error recovery as well as simple error detection, which are two separate issues, and should be covered by two separate tests.

## The Pascal acceptance test

The Pascal acceptance test consists of program constructs that are correct for ISO 7185 Pascal. It should both compile and run under a ISO 7185 Pascal compatible compiler. The acceptance test is present in a single source file:

iso7185pat.pas

## The Pascal rejection test

The Pascal rejection test is arranged as a series of short tests because each test is designed to fail. Thus, ideally only one failure point at a time is executed.

The test script, runprt, is designed so that each result is checked for if an error occurred or not, with the lack of an error being a fail. However, there are at least two characteristics of a test result that go to quality, and thus imply that the result should be hand checked at least once:

1. The indicated error may or may not be appropriate for the error that was caused (for example, indicating a missing ':' when the problem was a missing ';').
2. The error may generate many collateral errors, or even refuse to compile the remainder of the program. This includes generating recycle balance errors.

The error signature of the compile can be compared to the previous runs, but a miscompare does not automatically fail the compiler, since error handling may have changed. The typical stragey is to run a new compiler version through the PRT test, and hand-examine only the error differences. If the changes in the compiler are minor, a compare go/no go will serve as a regression test.

Here are descriptions of each of the tests.

Class 1: Syntax graph visitation. These tests are for various points on the syntax graph, typically addition or elidation of a critical symbol. These tests are not meant to be exhaustive, since there are an infinite number of possible syntax constructions. Rather, it is designed to be representative.

The syntax visitation is according to the Pascal Users's manual and Report" 4th edition, Appendix D, Syntax diagrams. The tests run in reverse, starting with "program", and working backwards. Thus, the syntax checks are performed top to bottom.

Class 2: Semantic tests. These tests are for the meaning of the program. Examples include using an undefined variable, using a type in the incorrect context, executing a case statement where there is no case matching the selector, etc.

Note that the tests are not designed to be a complete set of failure modes for the compiler, nor is that possible, since the set of possible failure modes is infinite. Instead, this test is designed to be representative", or contain a series of known failure modes that test the quality of error checking and recovery in the compiler.

Note that the semantic errors are the same as those enumerated in the ISO 7185 standard in Appendix D, offset in number by the section 1700. Not all errors in this section have equivalent test cases. It is possible for an error defined by the standard to be without expression as a program. The description of the error is as was written in the ISO 7185 standard.

The ISO 7158 listed errors are a mix of compile time and runtime errors. On some errors, either mode is possible, on others, not. For these cases, we have tried to break the test program into two sections, one that could fail at compile time, and others that do not. These are labeled with an appended A, B, C, etc.

There can be other reasons to divide a test case into two programs.

With semantic errors, there is always the posibility that the code could be changed by an agressive compiler to eliminate or nulify the test case. There is no requirement in ISO 7185 that the compiler render into code a statement that clearly has no effect in the final program, just to achieve an error result. Its really not possible to defeat this kind of optimization entirely. Printing to output is one possibility, but even then there is nothing to prevent the compiler from assuming it knows what the output will be and changing it. This is perhaps a subject for further research.

### List of tests

#### Class 1: Syntatic errors

Program

iso7185prt0001 Missing semicolon after program statement

iso7185prt0002 missing "program" word-symbol

iso7185prt0003 missing program name word-symbol

iso7185prt0004 Moved

iso7185prt0005 Moved

iso7185prt0006 missing period

iso7185prt0007 Extra semicolon

iso7185prt0008 Missing header parameters

iso7185prt0009 Opening paren for program header only

iso7185prt0010 Closing paren for program header only

iso7185prt0011 Improperly terminated header list

iso7185prt0012 Consecutive semicolons

Block

iso7185prt0013 Missing number for label

iso7185prt0014 Missing semicolon after label

iso7185prt0015 Non-numeric label

iso7185prt0016 Unterminated label list

iso7185prt0017 Unstarted label list

iso7185prt0018 Missing constant

iso7185prt0019 Missing constant right side

iso7185prt0020 Missing "=" in const

iso7185prt0021 Incomplete second in const

iso7185prt0022 Missing ident in const

iso7185prt0023 Missing semicolon in const

iso7185prt0024 Reverse order between label and const

iso7185prt0025 Missing type

iso7185prt0026 Missing type right side

iso7185prt0027 Missing "=" in type

iso7185prt0028 Missing ident in type

iso7185prt0029 Incomplete second in type

iso7185prt0030 Missing semicolon in type

iso7185prt0031 Reverse order between const and type

iso7185prt0032 Missing var

iso7185prt0033 Missing var right side

iso7185prt0034 Missing var ident list prime

iso7185prt0035 Missing var ident list follow

iso7185prt0036 Missing ":" in var

iso7185prt0037 Missing ident list in var

iso7185prt0038 Incomplete second in var

iso7185prt0039 Missing semicolon in var

iso7185prt0040 Reverse order between type and var

iso7185prt0041 Missing "procedure" or "function"

iso7185prt0042 Missing ident

iso7185prt0043 Missing semicolon

iso7185prt0044 Consecutive semicolons start

iso7185prt0045 Missing block

iso7185prt0046 Missing final semicolon

iso7185prt0047 Misspelled directive

iso7185prt0048 Bad directive

iso7185prt0049 Misspelled procedure

iso7185prt0050 Misspelled function

iso7185prt0051 Bad procedure/function

iso7185prt0052 Missing ":" on return type for function

iso7185prt0053 Missing type id on return type for function

iso7185prt0054 Reverse order between var and procedure

iso7185prt0055 Reverse order between var and function

iso7185prt0056 missing begin word-symbol

iso7185prt0057 missing end word-symbol

Statement

iso7185prt0100 Missing label ident

iso7185prt0101 Missing label ":"

iso7185prt0102 Missing assignment left side

iso7185prt0103 Missing assignment ":="

iso7185prt0104 Missing procedure identifier

iso7185prt0105 Missing "begin" on statement block

iso7185prt0106 Misspelled "begin" on statement block

iso7185prt0107 Missing "end" on statement block

iso7185prt0108 Mispelled "end" on statement block

iso7185prt0109 Missing "if" on conditional

iso7185prt0110 Misspelled "if" on conditional

iso7185prt0111 Missing expression on conditional

iso7185prt0112 Missing "then" on conditional

iso7185prt0113 Misspelled "then" on conditional

iso7185prt0114 Misspelled "else" on conditional

iso7185prt0115 Missing "case" on case statement

iso7185prt0116 Misspelled "case" on case statement

iso7185prt0117 Missing expression on case statement

iso7185prt0118 Missing "of" on case statement

iso7185prt0119 Misspelled "of" on case statement

iso7185prt0120 Missing constant on case stament list

iso7185prt0121 Missing 2nd constant on case statement list

iso7185prt0122 Missing ":" before statement on case statement

iso7185prt0123 Missing ";" between statements on case statement

iso7185prt0124 Missing "end" on case statement

iso7185prt0125 Misspelled "end" on case statement

iso7185prt0126 Missing "while" on while statement

iso7185prt0127 Mispelled "while" on while statement

iso7185prt0128 Missing expression on while statement

iso7185prt0129 Missing "do" on while statement

iso7185prt0130 Missing "repeat" on repeat statement

iso7185prt0131 Misspelled "repeat" on repeat statement

iso7185prt0132 Missing "until" on repeat statement

iso7185prt0133 Misspelled "until" on repeat statement

iso7185prt0134 Missing expression on repeat statement

iso7185prt0135 Missing "for" on for statement

iso7185prt0136 Misspelled "for" on for statement

iso7185prt0137 Missing variable ident on for statement

iso7185prt0138 Misspelled variable ident on for statement

iso7185prt0139 Missing ":=" on for statement

iso7185prt0140 Missing start expression on for statement

iso7185prt0141 Missing "to"/"downto" on for statement

iso7185prt0142 Misspelled "to" on for statement

iso7185prt0143 Misspelled "downto" on for statement

iso7185prt0144 Missing end expression on for statement

iso7185prt0145 Missing "do" on for statement

iso7185prt0146 Misspelled "do" on for statement

iso7185prt0147 Missing "with" on with statement

iso7185prt0148 Misspelled "with" on with statement

iso7185prt0149 Missing first variable in with statement list

iso7185prt0150 Missing second variable in with statement list

iso7185prt0151 Missing "goto" in goto statement

iso7185prt0152 Misspelled "goto" in goto statement

iso7185prt0153 Missing unsigned integer in goto statement

iso7185prt0154 Missing 1st constant on case statement list

iso7185prt0155 Missing only variable in with statement list

iso7185prt0156 Missing ',' between case constants

iso7185prt0157 Missing ',' between variables in with statement

Field list

iso7185prt0200 Missing field ident

iso7185prt0201 Missing first field ident

iso7185prt0202 Missing second field ident

iso7185prt0203 Missing ':' between ident and type

iso7185prt0204 Missing type

iso7185prt0205 Missing ';' between successive fields

iso7185prt0206 Misspelled 'case' to variant

iso7185prt0207 Missing identifier for variant

iso7185prt0208 Missing type identifier with field identifier

iso7185prt0209 Missing type identifier without field identifier

iso7185prt0210 Missing 'of' on variant

iso7185prt0211 Misspelled 'of' on variant

iso7185prt0212 Missing case constant on variant

iso7185prt0213 Missing first constant on variant

iso7185prt0214 Missing second constant on variant

iso7185prt0215 Missing ':' on variant case

iso7185prt0216 Missing '(' on field list for variant

iso7185prt0217 Missing ')' on field list for variant

iso7185prt0218 Missing ';' between successive variant cases

iso7185prt0219 Attempt to define multiple variant sections

iso7185prt0220 Standard field specification in variant

iso7185prt0221 Missing ',' between first and second field idents

iso7185prt0222 Missing ',' between first and second field idents in variant

Procedure or Function Heading

iso7185prt0300 Missing 'procedure'

iso7185prt0301 Misspelled 'procedure'

iso7185prt0302 Missing procedure identifier

iso7185prt0303 Missing 'function'

iso7185prt0304 Misspelled 'function'

iso7185prt0305 Missing function identifier

iso7185prt0306 Missing type ident after ':' for function

Ordinal Type

iso7185prt0400 Missing '(' on enumeration

iso7185prt0401 Missing identifier on enumeration

iso7185prt0402 Missing 1st identifier on enumeration

iso7185prt0403 Missing 2nd identifier on enumeration

iso7185prt0404 Missing ')' on enumeration

iso7185prt0405 Missing 1st constant on subrange

iso7185prt0406 Missing '..' on subrange

iso7185prt0407 Missing 2nd constant on subrange

iso7185prt0408 Missing ',' between identifiers on enumeration

Type

iso7185prt0500 Missing type identifer after '^'

iso7185prt0501 Misspelled 'packed'

iso7185prt0502 Missing 'array'

iso7185prt0503 Missing '[' on array

iso7185prt0504 Missing index in array

iso7185prt0505 Missing first index in array

iso7185prt0506 Missing second index in array

iso7185prt0507 Missing ']' on array

iso7185prt0508 Missing index specification in array

iso7185prt0509 Missing 'of' on array

iso7185prt0510 Misspelled 'of' on array

iso7185prt0511 Missing type on array

iso7185prt0512 Missing 'file' or 'set' on file or set type

iso7185prt0513 Misspelled 'file' on file type

iso7185prt0514 Missing 'of' on file type

iso7185prt0515 Missing type on file type

iso7185prt0516 Misspelled 'set' on set type

iso7185prt0517 Missing 'of' on set type

iso7185prt0518 Missing type on set type

iso7185prt0519 Missing 'record' on field list

iso7185prt0520 Misspelled 'record' on field list

iso7185prt0521 Missing 'end' on field list

iso7185prt0522 Misspelled 'end' on field list

Formal Parameter List

iso7185prt0600 Missing parameter identifier

iso7185prt0601 Missing first parameter identifier

iso7185prt0602 Missing second parameter identifier

iso7185prt0603 Missing ',' between parameter identifiers

iso7185prt0604 Missing ':' on parameter specification

iso7185prt0605 Missing type identifier on parameter specification

iso7185prt0606 Missing parameter specification after 'var'

iso7185prt0607 Misspelled 'var'

iso7185prt0608 Missing ';' between parameter specifications

Expression

iso7185prt0700 Missing operator

iso7185prt0701 Missing first operand to '='

iso7185prt0702 Missing second operand to '='

iso7185prt0703 Missing first operand to '>'

iso7185prt0704 Missing second operand to '>'

iso7185prt0705 Missing first operand to '<'

iso7185prt0706 Missing second operand to '<'

iso7185prt0707 Missing first operand to '<>'

iso7185prt0708 Missing second operand to '<>'

iso7185prt0709 Missing first operand to '<='

iso7185prt0710 Missing second operand to '<='

iso7185prt0711 Missing first operand to '>='

iso7185prt0712 Missing second operand to '>='

iso7185prt0713 Missing second operand to 'in'

iso7185prt0714 Alternate '><'

iso7185prt0715 Alternate '=<'

iso7185prt0716 Alternate '=>'

iso7185prt0717 Missing first operand to 'in'

Actual Parameter List

iso7185prt0800 Empty list (parens only)

iso7185prt0801 Missing leading '(' in list

iso7185prt0802 Missing ',' in parameter list

iso7185prt0803 Missing first parameter in parameter list

iso7185prt0804 Missing second parameter in parameter list

iso7185prt0805 Missing ')' in list

Write Parameter List

iso7185prt0900 Empty list (parens only)

iso7185prt0901 Missing leading '(' in list

iso7185prt0902 Missing ',' in parameter list

iso7185prt0903 Missing first parameter in parameter list

iso7185prt0904 Missing second parameter in parameter list

iso7185prt0905 Field with missing value

iso7185prt0906 Fraction with missing value

iso7185prt0907 Field and fraction with missing field

iso7185prt0908 Missing ')' in list

Factor

iso7185prt1000 Missing leading '(' for subexpression

iso7185prt1001 Missing subexpression in '()'

iso7185prt1002 Misspelled 'not'

iso7185prt1003 'not' missing expression

iso7185prt1004 Missing '[' on set constant

iso7185prt1006 Missing first expression in range

iso7185prt1007 Missing second expression in range

iso7185prt1008 Missing '..' or ',' in set constant list

iso7185prt1009 Missing first expression in ',' delimited set constant list

iso7185prt1010 Missing second expression in ',' delimited set constant list

Term

iso7185prt1100 Missing first operand to '\*'

iso7185prt1101 Missing second operand to '\*'

iso7185prt1102 Missing first operand to '/'

iso7185prt1103 Missing second operand to '/'

iso7185prt1104 Missing first operand to 'div'

iso7185prt1105 Missing second operand to 'div'

iso7185prt1106 Missing first operand to 'mod'

iso7185prt1107 Missing second operand to 'mod'

iso7185prt1108 Missing first operand to 'and'

iso7185prt1109 Missing second operand to 'and'

Simple Expression

iso7185prt1200 '+' with missing term

iso7185prt1201 '-' with missing term

iso7185prt1203 Missing second operand to '+'

iso7185prt1205 Missing second operand to '-'

iso7185prt1206 Missing first operand to 'or'

iso7185prt1207 Missing second operand to 'or'

Unsigned Constant

iso7185prt1300 Misspelled 'nil'

Variable

iso7185prt1400 Missing variable or field identifier

iso7185prt1401 Missing '[' in index list

iso7185prt1402 Missing expression in index list

iso7185prt1403 Missing first expression in index list

iso7185prt1404 Missing second expression in index list

iso7185prt1405 Missing ',' in index list

iso7185prt1406 Missing ']' in index list

iso7185prt1407 Missing field identifier after '.'

Unsigned number

iso7185prt1500 Missing leading digit before '.'

iso7185prt1501 Missing digit after '.'

iso7185prt1502 Missing digit before and after '.'

iso7185prt1503 Mispelled 'e' in exponent

iso7185prt1504 Missing 'e' in exponent

iso7185prt1505 Missing digits in exponent

iso7185prt1506 Missing digits in exponent after '+'

iso7185prt1507 Missing digits in exponent after '-'

iso7185prt1508 Missing number before exponent

Character String

iso7185prt1600 String extends beyond eol (no end quote)

#### Class 2: Semantic errors

iso7185prt1701 For an indexed-variable closest-containing a single index-expression, it is an error if the value of the index-expression is not assignment-compatible with the index-type of the array-type.

iso7185prt1702 It is an error unless a variant is active for the entirety of each reference and access to each component of the variant.

iso7185prt1703 It is an error if the pointer-variable of an identified-variable denotes a nil-value.

iso7185prt1704 It is an error if the pointer-variable of an identified-variable is undefined.

iso7185prt1705 It is an error to remove from its pointer-type the identifying-value of an identified-variable when a reference to the identified-variable exists.

iso7185prt1706 It is an error to alter the value of a file-variable f when a reference to the buffer-variable f^ exists.

iso7185prt1707 It is an error if the value of each corresponding actual value parameter is not assignment compatible with the type possessed by the formal-parameter.

iso7185prt1708 For a value parameter, it is an error if the actual-parameter is an expression of a set-type whose value is not assignment-compatible with the type possessed by the formal-parameter.

iso7185prt1709 It is an error if the file mode is not Generation immediately prior to any use of put, write, writeln or page.

iso7185prt1710 It is an error if the file is undefined immediately prior to any use of put, write, writeln or page.

iso7185prt1711 It is an error if end-of-file is not true immediately prior to any use of put, write, writeln or page.

iso7185prt1712 It is an error if the buffer-variable is undefined immediately prior to any use of put.

iso7185prt1713 It is an error if the file is undefined immediately prior to any use of reset.

iso7185prt1714 It is an error if the file mode is not Inspection immediately prior to any use of get or read.

iso7185prt1715 It is an error if the file is undefined immediately prior to any use of get or read.

iso7185prt1716 It is an error if end-of-file is true immediately prior to any use of get or read.

iso7185prt1717 For read, it is an error if the value possessed by the buffer-variable is not assignmentcompatible with the variable-access.

iso7185prt1718 For write, it is an error if the value possessed by the expression is not assignment-compatible with the buffer-variable.

iso7185prt1719 For new(p,c l ,...,c n,), it is an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

iso7185prt1720 For dispose(p), it is an error if the identifying-value had been created using the form new(p,c l ,...,c n ).

iso7185prt1721 For dispose(p,k l ,...,k, ), it is an error unless the variable had been created using the form new(p,c l ,...,c,,,) and m is equal to n.

iso7185prt1722 For dispose(p,k l ,...,k, ), it is an error if the variants in the variable identified by the pointer value of p are different from those specified by the case-constants k l ,...,k,,,,.

iso7185prt1723 For dispose, it is an error if the parameter of a pointer-type has a nil-value.

iso7185prt1724 For dispose, it is an error if the parameter of a pointer-type is undefined.

iso7185prt1725 It is an error if a variable created using the second form of new is accessed by the identified variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

iso7185prt1726 For pack, it is an error if the parameter of ordinal-type is not assignment-compatible with the index-type of the unpacked array parameter.

iso7185prt1727 For pack, it is an error if any of the components of the unpacked array are both undefined and accessed.

iso7185prt1728 For pack, it is an error if the index-type of the unpacked array is exceeded.

iso7185prt1729 For unpack, it is an error if the parameter of ordinal-type is not assignment-compatible with the index-type of the unpacked array parameter.

iso7185prt1730 For unpack, it is an error if any of the components of the packed array are undefined.

iso7185prt1731 For unpack, it is an error if the index-type of the unpacked array is exceeded.

iso7185prt1732 Sqr(x) computes the square of x. It is an error if such a value does not exist.

iso7185prt1733 For ln(x), it is an error if x is not greater than zero.

iso7185prt1734 For sqrt(x), it is an error if x is negative.

iso7185prt1735 For trunc(x), the value of trunc(x) is such that if x is positive or zero then 0 < x-trunc(x) < 1; otherwise 1 <x-trunc(x) < 0. It is an error if such a value does not exist.

iso7185prt1736 For round(x), if x is positive or zero then round(x) is equivalent to trunc(x+0.5), otherwise round(x) is equivalent to trunc(x- 0.5). It is an error if such a value does not exist.

iso7185prt1737 For chr(x), the function returns a result of char-type that is the value whose ordinal number is equal to the value of the expression x if such a character value exists. It is an error if such a character value does not exist.

iso7185prt1738 For succ(x), the function yields a value whose ordinal number is one greater than that of x, if such a value exists. It is an error if such a value does not exist.

iso7185prt1739 For pred(x), the function yields a value whose ordinal number is one less than that of x, if such a value exists. It is an error if such a value does not exist.

iso7185prt1740 When eof(f) is activated, it is an error if f is undefined.

iso7185prt1741 When eoln(f) is activated, it is an error if f is undefined.

iso7185prt1742 When eoln(f) is activated, it is an error if eof(f) is true.

iso7185prt1743 An expression denotes a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use is an error.

iso7185prt1744 A term of the form x/y is an error if y is zero.

iso7185prt1745 A term of the form i div j is an error if j is zero.

iso7185prt1746 A term of the form i mod j is an error if j is zero or negative.

iso7185prt1747 It is an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

iso7185prt1748 It is an error if the result of an activation of a function is undefined upon completion of the algorithm of the activation.

iso7185prt1749 For an assignment-statement, it is an error if the expression is of an ordinal-type whose value is not assignment-compatible with the type possessed by the variable or function-identifier.

iso7185prt1750 For an assignment-statement, it is an error if the expression is of a set-type whose value is not assignment-compatible with the type possessed by the variable.

iso7185prt1751 For a case-statement, it is an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

iso7185prt1752 For a for-statement, it is an error if the value of the initial-value is not assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

iso7185prt1753 For a for-statement, it is an error if the value of the final-value is not assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

iso7185prt1754 On reading an integer from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-integer.

iso7185prt1755 On reading an integer from a textfile, it is an error if the value of the signed-integer read is not assignment-compatible with the type possessed by variable-access.

iso7185prt1756 On reading a number from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-number.

iso7185prt1757 It is an error if the buffer-variable is undefined immediately prior to any use of read.

iso7185prt1758 On writing to a textfile, the values of TotalWidth and FracDigits are greater than or equal to one ; it is an error if either value is less than one.

iso7185prt1759 The execution of any action, operation, or function, defined to operate on a variable, is an error if the variable is a program-parameter and, as a result of the binding of the program-parameter, the execution cannot be completed as defined.

#### Class 3: Advanced error checking

These are advanced error checks that the compiler may or may not check for.

iso7185prt1800 Access to dynamic variable after dispose.

#### Class 4: Field checks

These tests are a collection of issues found while running the compiler.

iso7185prt1900 Access to dynamic variable after dispose.

### Running the PRT and interpreting the results

There is a single script that runs the prt:

$ runprt

The results of all of the tests are gathered and formatted into the file:

iso7185prt.lst

This file has several sections:

1. List of tests with no compile or runtime error.
2. List of differences between compiler output and “gold” standard outputs.
3. List of differences between runtime output and “gold” standard outputs.
4. Collected compiler listings and runtime output of all tests.

Each of these will be examined in turn.

#### List of tests with no compile or runtime error.

The purpose of the rejection test is to cause an error in the test program, either at compile time or runtime, and test its proper handling. By definition, if no error exists, the test has failed. This list gives the tests that need to be examined for why they didn’t detect an error.

#### List of differences between compiler output and “gold” standard outputs.

Shows the number of lines difference between the output test.err or compiler output listing, and test.ecp or “gold standard” compiler output listing, for each test program. A difference of 0 means that nothing has changed in the run.

The compiler output listing flagging an error and the run output flagging an error are mutually exclusive. If the compiler listing shows an error, the program will not be run. If if the compiler output shows an error, it is examined to see if it flagged an appropriate error for the fault, and that it didn’t generate excessive “cascade” or further errors caused by the compiler having difficulty recovering from the error.

Once the test.err file is judged satisfactory, it is copied to the test.ecp file as the “gold” standard. If the test run shows a difference between the current compile and the gold standard file, it does not necessarily mean that it is wrong. It simply means it needs to be reevaluated and perhaps copied as the new gold standard file.

#### List of differences between runtime output and “gold” standard outputs.

If the test file is successfully compiled, it is run and the output collected as test.lst. This is compared to the “gold” run output file test.cmp.

The run output file, if it exists, is checked to see if it indicates an appropriate error for the fault indicated in the test. If the run indicates no error, or an error that is not related to the fault, or simply crashes, that is a failure to properly handle the fault.

Once the test.lst file is judged satisfactory, it is copied to the test.cmp file as the “gold” standard. If the test run shows a difference between the current output and the gold standard file, it does not necessarily mean that it is wrong. It simply means it needs to be reevaluated and perhaps copied as the new gold standard file.

#### Collected compiler listings and runtime output of all tests.

The output of the compiler listing, as well as the output of the run, if that exists, for each test is concatenated and placed at the end of the listing file. This allows the complete output of each test to be examined, without having to look at the individual test.err or test.list file.

### Overall interpretation of PRT results

The rejection test is a “quality” test. There is no absolute right and wrong. The clearest indication of failure is failure to find any error. Past that are issues that indicate the quality of the error handling in the compiler:

1. If the error is indicative of what the failure was.
2. No or few further errors resulted after the failure.
3. The compiler was able to resyncronise with the source.
4. No data was corrupted as a result of the run (dynamic space imbalance, null pointer errors or similar).

You can say that failure to achieve the above marks a poor quality compiler, but not a failing test.

## Sample program tests

The sample program tests give a series of sample programs in Pascal. The idea of the sample programs tests is to prove out operation on common programs, and also to give a newly modified version of P6 a series if tests progressing from the most simple (“hello world”) to more complex tests. If the new version of the compiler has serious problems, it is better to find out with simple tests rather than have it fail on a more complex, and difficult to debug, test.

Hello Gives the standard “hello, world” minimum test.

Roman Prints a series of roman numerals. From Jensen and Wirth’s “User Manual and report”.

Match A number match game, this version from Conway, Gries and Zimmerman “A primer on Pascal”.

Startrek Plays text mode startrek. From the internet.

Basics Runs a subset Basic interpreter. It is tested by running a recoded version of “match” above.

Pascal-S Runs a subset of ISO 7185 Pascal. From Niklaus Wirth’s work at ETH. It is tested by running the “Roman” program above.

## Previous Pascal-P versions test

As part of the regression tests, Pascal-P6 runs the older versions of itself, namely Pascal-P2 and Pascal-P4. These are the only versions of the compiler available. See the section “introduction” on page 7, and also the historical material on Pascal-P on the Standard Pascal website.

The run of previous versions of Pascal-P perhaps constitutes the purest form of regression test. It not only insures that P6 is compatible with previous versions, but that it can actually compile and run all of the previous code. Of course, this is possible in main because these 1970’s versions were adapted to the ISO 7185 standard, but that, fortunately, was a small change.

### Compile and run Pascal-P2

P2 runs on P6 virtually unmodified. It needed only the declarations of the prd and prr files removed. This is required because they are predefined in P6. In a stand-alone use of P2, those files are real external files and have to be declared.

P2 runs as a test a modified version of roman.pas. It has to be modified because of:

1. P2 only supports upper case.
2. P2 does not support the output file as a default parameter to writeln.

These changes are documented in the P2 archive on <http://www.standardpascal.org>.

P2 does not have a full regression test of its own because I didn’t do the work of cutting down the ISO 7185 test to the subset that P2 implements, as I did for P4. This is perhaps a future project.

### Compile and run Pascal-P4

P4 has the same modification of the prd and prr files as P2, but otherwise runs unmodified. P4 runs as its target standardp.pas, which was a version of iso7185pat.pas that was stripped so as to fit the subset of ISO 7185 that P4 runs. Recall that P4 does not run any standard of Pascal at all, it is intentionally a subset.

Standard.pas was modified only in that P4 running under P6 cannot execute a for loop of the form:

for b := false to true do ...

The reason for this is non-trival. The way that P4 executes such a loop is to check that b lies in the range of boolean 0..1, and terminate if it lies outside that range. This actually makes it an illegal program if variable b is treated as boolean, since b will be incremented to 2 and thus be an out of range value.

This is actually a standard issue with Pascal in general, and the ISO 7185 standard took pains to show an equivalent form of the for loop that does not involve creating an out of range boolean. P6 also uses a form internally that fixes this, it has to in order to fully comply with the ISO 1785 standard.

So why does that work with a standalone P4 and not with a P4 running under P6? P4, as well as P6, treat the internal data store as typeless and use type escapes in the form of undiscriminated variants in order to differentiate the different data forms. This means that the result of assigning 2 to b (or incrementing it from 1) are system and compiler dependent. Thus, the direct code generating compilers tolerate this, and P6 simply does not.

## Self compile

One of the more difficult tests for P6 (and the most time consuming) is to have P6 compile and run itself. Actually the entire idea of Pascal-P was to compile and run itself in order to accomplish a bootstrap of the compiler. Pascal-P was never provided in a form able to directly compile itself, it needed a few modifications in order to do that. Also, self compiling a compiler that interprets its final code is different from a machine code generating compiler. Whereas it is conceptually simple to imagine the parser and intermediate code generator compling a version of itself, having the interpreter run another instance of itself on itself is like looking into a series of mirrors. The interpreter will be interpreting itself while that interpreter interprets another program, etc.

This can actually be carried out to any depth of self-interpretation, but because each interpretation level slows down the code by an order of magnitude, such nested self interpretation rapidly becomes impractical to complete. As it is, a single level of self-interpretation takes hours.

The reason to put work into a self compile of P6 is that it is a difficult test that goes a long way to prove out the stability and capacity of the compiler, and also because it proves out a very important function of Pascal-P, that of bootstrapping itself.

### pcom

I was able to get pcom.pas to self compile. This means to compile and run pcom.pas, then execute it in the simulator, pint. Then it is fed its own source, and compiles itself into intermediate code. Then this is compared to the same intermediate code for pcom as output by the regular compiler. Its a good self check, and in fact found a few bugs.

The Windows batch file to control a self compile and check is:

cpcoms.bat

What does it mean to self compile? For pcom, not much. Since it does not execute itself (pint does that), it is simply operating on the interpreter, and happens to be compiling a copy of itself.

#### Changes required

pcom won't directly compile itself the way it is written. The reason is that the "prr" file, the predefined file it uses to represent it's output file is only predefined to P6 itself. The rules for ISO 7185 are that each file that is defined in the header which is not predefined, such as "input" or "output", must be also declared in a var statement. This makes sense, because if it is not a predefined file, the compiler must know what type of file (or even non-file) that is being accessed externally to the program.

Because the requirements are different from a predefined special compiler file to a file that is simply external, the source code must be different for a P6 file vs. another compiler. A regular ISO 7185 Pascal compiler isn't going to have a predefined prr file.

The change is actually quite small, and marked in the source. You simply need to remove, or comment out, the following statements:

{ !!! remove this statement for self compile }

prr: text;                      { output code file }

and

{ !!! remove this statement for self compile }

rewrite(prr); { open output file }

In the batch file above, this modified file is represented as pcomm.pas, or "modified" pcom.pas.

All of the source code changes from pcom.pas to pcomm.pas are automated in cpints.bat.

### pint

Pint is more interesting to self compile, since it is running (being interpreted) on a copy of itself. Unlike the pcom self compile, pint can run a copy of itself running a copy of itself, etc., to any depth. Of course, each time the interpreter runs on itself, it slows down orders of magnitude, so it does not take many levels to make it virtually impossible to run to completion. Ran a copy of pint running on itself, then interpreting a copy of iso7185pat. The result of the iso7185pat is then compared to the "gold" standard file.

As with pcom, pint will not self compile without modification. It has the same issue with predefined header files. Also, pint cannot run on itself unless its storage requirements are reduced. For example, if the "store" array, the byte array that is used to contain the program, constants and variables, is 1 megabyte in length, the copy of pint that is hosted on pint must have a 1 megabyte store minus all of the overhead associated with pint itself.

The windows batch file required to self compile pint is:

cpints.bat

As a result, these are the changes required in pint:

{ !!! Need to use the small size memory to self compile, otherwise, by

  definition, pint cannot fit into its own memory. }

maxstr      = 2000000;  { maximum size of addressing for program/var }

{maxstr      = 200000;}  { maximum size of addressing for program/var }

and

{ !!! remove this next statement for self compile }

prd,prr     : text;(\*prd for read only, prr for write only \*)

and

{ !!! remove this next statement for self compile }

reset(prd);

and

{ !!! remove this next statement for self compile }

rewrite(prr);

All these changes were made in the file pintm.pas.

Pint also has to change the way it takes in input files. It cannot read the intermediate from the input file, because that is reserved for the program to be run. Instead, it reads the intermediate from the "prd" header file. The interpreted program can also use the same prd file. The solution is to "stack up" the intermediate files. The intermediate for pint itself appears first, followed by the file that is to run under that (iso7185pat). It works because the intermediate has a command that signals the end of the intermediate file, "q". The copy of pint that is reading the intermediate code for pint stops, then the interpreted copy of pint starts and reads in the other part of the file. This could, in fact, go to any depth.

All of the source code changes from pint.pas to pintm.pas are automated in cpints.bat.

Self compiled files and sizes

The resulting sizes of the self compiled files are:

pcomm.P6

Storage areas occupied

|  |  |  |
| --- | --- | --- |
| Contents | Range of storage | Net size |
| Program | 0-114657 | (114658) |
| Stack/Heap | 114658-1987994 | (1873337) |
| Constants | 1987995-2000000 | (  12005) |

pintm.P6

Storage areas occupied

|  |  |  |
| --- | --- | --- |
| Contents | Range of storage | Net size |
| Program | 0- 56194 | (56195) |
| Stack/Heap | 56195-1993985 | (1937791) |
| Constants | 1993986-2000000 | (6014) |

# pgen\_gcc\_i80386

pgen is a new package for P6. It gives the ability to generate code directly for target machines. It is designed for the following goals:

1. Use the existing P6 intermediate language with as few modifications as possible.
2. Able to convert to target different machines as easily as possible.

pgen versions have the following naming scheme:

pgen\_<C compiler>\_<machine>

The C compiler as the particular ISO/ANSI C compiler targeted, and the machine is the target machine.

At this writing, only GCC as a the compiler, and the 32 bit I80386 machine model are available as a target.

pgen generates an assembly language output file, compatible with the C compiler target. To result in a full compilation, it requires:

1. A Pascal to clib translator/support “thunk” package, written in ISO/ANSI C, and using the clib standard C support routines package.
2. A Pascal runtime support package, written in the target assembly language.

Together with the generated assembly language file, these packages result in a binary that can run on any ISO/ANS C compatible installation.

## Generated assembly file conventions

The C language support file for P6 can run on any ISO/ANSI C installation. However, assembly language conventions are far less standardized. Thus, pgen must produce assembly language output specific to the assembler that comes with the C compiler in use.

For GCC, GAS or “GNU Assembler” is used, specifically for the 32 bit I80386 assembler. It follows the AT&T assembly syntax, notable for the following conventions:

1. Sources are on the left, and destinations are on the right.
2. Each operator contains the size of the operation, byte, 16 bit word, 32 bit double word, and 64 bit quad word as an operator suffix (b, w, l or q).

This is not the complete list of characteristics. Please see the GAS user’s manual for further details.

The generated assembly file contains three distinct areas:

1. The generated code.
2. The global (uninitialized) space.
3. The constants area.

In Pascal, there is no global initialized, but writable space. Similarly, the constants area cannot be written. The global space is where all program block variables reside, and is allocated in the file by a single block definition. The constants area contains a series of constants defined in the target assembly language, and are prefaced by labels for their addresses.

In Pascal, the module order is for the program block to appear last in the file. It is prefaced by the label “\_pascal\_main”, and it is called by the C support module.

## Labeling conventions

All labels in the P6 binary system are prefaced with “\_pascal\_”. This prevents collision with the C language names in the source. Most labels used in the binarieshave no symbolic meaning and are numbered as in “\_pascal\_42”.

The exception to this rule are the direct support library calls. These have specific names.

The original symbols from the P6 source files are not carried in the P6 intermediate, and do not appear in the output.

Each input line is output to the assembly output file in the format:

// line: 42

This can be used to find what code was generated for a particular line in the source file.

## Calling conventions

pgen uses the GCC C calling convention. That is, the register based calling convention that GCC uses when one C function calls another. It does not use the system calling convention or any other convention in the target program. All calls to the operating system, or to alternate languages, are left to the support module.

There is no direct calling convention between Pascal and C. Although the GCC calling convention is used, there are several places where the parameters in C have no analog in Pascal. These include (but are not limited to):

* Void pointers
* Pointers that are interchangeable with arrays
* Pointers that can also be numbers, such as small integers that cannot be addresses
* Array “pointers” that can be null (0).
* Zero terminated strings.

However, all current P6 parameter data types ***can*** be described in C. Thus, all calls from Pascal to C functions must use a “thunk”, or short section of code that rearranges or filters the data to be correct for it’s C counterpart. The clib interface for the support library performs this for standard P6 support calls. This only leaves “external” calls from Pascal to C.

Please note that many Pascal implementations that “claim” to be able to directly call C from Pascal do this by incorporating many C language conventions into Pascal, and usually do that by breaking the type security of Pascal. P6 does not use this methodology. Instead, the same effect is done by creating a thunk, and that achieves the same end with full type security and little overhead.

To aid in the writing of thunks, P6 uses the rule that the length of arrays are always passed with an array as a prefix parameter. This means that the call:

type a = array [1..10] of integer;

procedure x(p: a); external;

Will be equivalent to ISO/ANSI C:

Void x(int al, int p[]);

With al being the lenth of the array p, which in this case will always be 10 by Pascal rules. In this way, the C code can always understand the length of the array passed without further Pascal aid.

## clib interface library and support module

The clib thunk layer and support library does the job of translating P6 I/O, memory management, and other support calls to the ISO/ANSI compatable clib support library that implements C support calls. The use of clib means that P6 can be hosted on any ISO/ANSI C compatible installation. It also means that this layer is automatically compatible with clib compatable systems such as Petit-ami.

The pascal\_support library also contains the main (or \_main) procedure that is the standard entry point for C. It contains a short entry and exit sequence, and then directly calls \_pascal\_main, which is the Pascal program block.

P6 generates a series of calls to the support library as follows:

void \_pascal\_get(file f);

void \_pascal\_put(file f);

void \_pascal\_readln(file f);

void \_pascal

…

The API for P6 support functions follows the IP Pascal support library API convention.

## Modifications required to pcom for pgen

Pgen uses the following layout for the target program:

Stack

...

...

Heap

Global variables

Constants

00000000: Program code

The heap grows upwards, and the stack grows downward. This is the opposite of the pint convention. For this reason pgen requires the “a+” or “alternate stack addressing” option be set in pcom when generating intermediates for pgen.

## Overview of pgen\_gcc\_i80386 functionality

Pgen starts by assembling and processing the intermediate much as pint does. The main difference is that it generates actual machine code, not pseudo-machine code, and outputs it to an assembler file instead of storing it to a code array. It outputs the file as text line instructions to an assembler instead of binary codes. The procedure assemble does this work in pgen, just as it does in pint. It reads the lines in the intermediate file, converts the codes to intermediate codes, but uses these to directly issue instructions to write the equivalent instructions to perform the function of the intermediate code to the output file.

The intermediate codes, as well as the basic order of processing used in the main procedure of pint, were maintained in pgen to make it easier to read and compare to pint functionality, as well as match the intermediate code documentation.

The key routine in assemble for generating instructions is wrtins. This routine takes a string and four parameters, two numerics parameters and two registers, and uses printf() style escape characters to format output code lines. For each call to wrtins, one line of output assembly code is generated. The parameter escapes are:

#0 or #1 Generates one of the numeric parameters, either the first or second one.

%r1 or %r2 Generates one of the register parameters, either the first or second one.

Note that the appearance of the escaped parameters is very much like the ones used in the GAS assembly language. This means that the strings used to form instructions look like their GAS equivalents.

And example of wrtins use is:

Wrtins(prr, ‘ mov #0($r1),%r2 ‘, 10, 0, ord(rgrax), ord(rgrax));

Would generate the output line:

mov #10(%eax),%eax

i.e., move the dword at offset 10 from eax to eax itself.

pgen is a register directed compiler, that is, it places all operands in registers and operates on them there. This extends even to real values, set values, structures and arrays. If the operand is too large to be placed in a dword register, then it is pushed onto the stack and a pointer created in a register that points to that. When the operand is removed from the stack, the stack is incremented over the object, and the register pointer discarded.

It does not use the floating point hardware stack feature of the I80386 floating point processor. Instead, all floating point values are pulled to the stack, and when it is time to operate on them, they are loaded into the FPU, and the result pushed back on the stack. This means that we don’t need to worry about fill spill handlers for the FPU, or preserving its registers.

### Processing expressions

All expressions, when found in the intermediate, are loaded into memory as expression trees. This is a compromise between fully graphing the intermediate and processing it as a linear sequence, as pint does. This method was used in the original PDP-11 C compiler and is common. It means that only single expression is stored in memory, the format of it is quite simple, and yet fairly good register optimization can be used. A tree traversal is quite simple.

The processing of trees separates the participating intermediates into loads, operators and stores. The loads are the leaves of the tree, the operators its branches, and the stores are the root of the tree.

Keep in mind also that there are a lot of optimizations made possible by expression tree formatting that are not yet implemented in pgen, such as common subexpression elimination, constant compression, etc.

### Register allocation

pgen uses a subset of SSA (Single Static Assignment) techniques I have termed here as “Virtual Register Allocation”. The idea of VRA is that a series of “Virtual Registers” from an unlimited set are assigned to each unique path in an expression that carries a value. These paths are then repeatedly evaluated to see if the same registers can be reused without conflict between multiple paths, and merged if there is no conflict. When and if the number of virtual registers is less than or equal to the number of physical registers, each VR is assigned a PR (Physical register), and the algoritim is complete.

Each operator is classed as a producer or a consumer, that is, it produces a value or it consumes one. A load would be an example of a producer, and a store is an example of a consumer. An add, for example, is both a consumer and a producer, since it consumes its operands, but also produces as result.

The VRs assigned contain the PRs for the I80386 as the lowest 6 registers. The register algoritim starts by assigning one of N unique registers (or virtual registers) to each producer, starting with the PRs, which would normally be consumed by the operator that uses it, above it in the tree. However, because I80386 as a binary operand based instruction set, that is, it specifies one operand as source, and the other as both source and destination, we assign one of the source registers as also being the destination. This can either be because the I80386 instruction set requires it, or done arbitrarily (as in the case of an add). This means that the same register number can travel across any number of expression nodes.

Once all nodes in the exression have been “painted” with register numbers, there are then N “domains” which establish the use of the virtual registers. At some point, we will map the virtual registers to real registers that have equivalence in the I80386 machine model, or eax, ebx, ecx, edx, esi and edi (6 registers in total). We actually define the first 6 virtual registers to be equivalent to the physical registers in order of 1=eax, 2=ebx, 3=ecx, 4=edx, 5=esi, and 6=edi. The goal will be to reduce the number of virtual registers to less than or equal to 6. At that time, we would be done, since the VRs are all then represented by PRs.

Because the I80386 machine requires the use of certain registers, we have to provide for fixed allocation for certain nodes. When we assign VRs to nodes, we allow each node to choose a specific PR. It assigns VRs to all of the other used registers that don’t have that requirement. We handle PR coflicts, for example, having a node that uses PR x have as one of its operands the same PR x by having two registers as the result, a result register and a transfer register. These are normally kept equal, but if the result of one node conflicts with a register used by the consumer node, it will be assigned a VR. When the tree is encoded, the fact that rr is not equal to rt is seen, and a move generated to push the result out of the way.

Note that even instruction sets that have no fixed physical allocations require fixed allocation processing. This is because registered parameters must be placed for calling convention purposes.

To reduce the number of VRs after allocation to equal or less than the number of PRs, we perform a (rather inefficient) search between all registers to determine if two VRs can be collapsed into one. We allow VRs to be merged into PRs, but not the converse, since that would break the fixed allocation of PRs done earlier. We kept track of the total Rs used. Then for each VR, we match all other Rs present and look for if they do or do not “intersect”, or have nodes that use both Rs. If they are found not to intersect, then we can merge them, which means finding all instances of the register to be merged and changing them to the target register number.

When we have performed all possible merges, we check if this has reduced the number of registers to less than or equal to PR. If so, we can stop there. All registers now bear PR numbers. This is actually the most common case. For an expression not to be able to so merge, it must be very complex, or use lots of fixed register allocations, or both. Nevertheless, we must provide a method to deal with it.

We remove these conflicts by the process of “pruning” the given expression trees. The top level node has its branches pruned off and made into freestanding trees one by one until the register allocation algorithim can process them properly. Each pruned branch is cleared of registers and reallocated, then an implicit store used to place the result of the partial tree in a local temp value. The place where it was pruned off gets a load of that same temp. If pruning the top node does not work, we recurs and try pruning the subtrees of it, and so forth, until we converge on an answer that works. However, this is a rare case as discussed above.

The register allocation algorithim discussed has lots of room for improvement, but is a good starting point, and is relatively easy to implement. It is also very good across multiple archiectures.

1. Many Pascal compilers that feature an ISO 7185 standards compliance flag in fact change the source language to be incompatible with ISO 7185 if the flag is off. Pascal-P6 does not do this. [↑](#footnote-ref-1)
2. The term “dialect” is the correct term for a variant of a language that is not close enough to be mutually understood between different speaking peoples or computer language implementations. [↑](#footnote-ref-2)
3. Although UCSD Pascal is no longer in wide use, that language had much of its constructs copied into other dialects, including Borland, Apple Pascal and many others. [↑](#footnote-ref-3)